Academiejaar 2013 - 2014

UNIVERSITEIT ANTWERPEN

FACULTEIT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

Gitaarmuziek Genereren Met Optimalisatietechnieken

Max Alexander Hein

Masterproef voorgedragen tot het bekomen
van de graad van:

Master in de Toegepaste Economische
Wetenschappen - Handelsingenieur

Promotor:

Drs. Dorien Herremans

Academiejaar 2013 - 2014

UNIVERSITEIT ANTWERPEN

FACULTEIT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

Gitaarmuziek Genereren Met Optimalisatietechnieken

Max Alexander Hein

Masterproef voorgedragen tot het bekomen
van de graad van:

Master in de Toegepaste Economische
Wetenschappen - Handelsingenieur

Promotor:

Drs. Dorien Herremans

Abstract

In deze thesis wordt er gezocht naar goede methodes om computergegenereerde muziek te maken. Het doel is om algoritmes te schrijven die neoklassieke gitaarsolo's kunnen creëren. De thesis beperkt zich tot die technieken die ook in de economische wetenschappen voorkomen. Concreet gaat het dan over stochastische Markov processen en metaheuristieken. Aangezien muziek wordt gezien als een optimalisatieprobleem, moet het ook als dusdanig uitgedrukt kunnen worden. Om tot deze uitdrukking te komen, worden er enkele benaderingsstrategieën gehanteerd. Vervolgens worden er een aantal experimenten gedaan om het project, namelijk het algoritmisch genereren van muziek, te realiseren. Dit brengt een leerproces met zich mee, waarbij er telkens voortgebouwd wordt op voorgaande ervaringen om steeds meer geavanceerde algoritmes te schrijven. Uiteindelijk resulteert het project in twee succesvolle algoritmes die volledige, aanhoorbare gitaarsolo's genereren aan de hand van gegevens die uit een referentiesolo worden gehaald. In beide algoritmes worden oplossingen gemaakt aan de hand van Markov ketens. Deze Markov ketens worden vervolgens iteratief geoptimaliseerd met behulp van heuristieken. In het ene algoritme gebeurt dit met Local Search, het andere algoritme is gebaseerd op de Ant Colony Optimization metaheuristiek.

Executive Summary

The central subject of this master thesis is computer aided composing of neoclassical guitar music. Neoclassical is a guitar style derived from baroque and romantic classical music, and is most prolific in terms of speed and technicality. It was developed around 1970, most notably by the hard rock band Deep Purple. Early examples of guitar players who adopted this style are Eddie Van Halen with his right hand tapping technique, and Ozzy Osbourne guitarist Randy Rhoads. In the eighties the neoclassical style became popular with many guitar virtuosos, and today it takes up a solid part of metal guitar soloing. The goal of this thesis is to create algorithms that generate guitar solos in this specific style.

This work is written by a master student in business engineering, an education similar to applied economics. Therefore, music will only be created with methods from the economic sciences. Above all, the focus will be on heuristic optimization techniques. To use heuristics for optimization problems, one needs to define a problem first. Hence, music as a whole will be viewed as a problem and generated music as a solution to that problem. Defining neoclassical guitar music as an economic problem is a difficult task, because music isn't a problem in its own way. A perspective must be formed to force music into a mold, where it can be treated like a problem. The point of view taken to create this mold not only defines the problem, put also a part of the solution. The fact that a framework is used, means that certain aspects of the problem are magnified, while other aspects are ignored. There are experiments in this work with different points of view to see which one is the most successful.

Another step that needs to be taken before a solution can be found, is looking into the methods used to create these solutions. Computer Aided Composing is a very divergent field, so only methods with a link to the specified musical genre and inspiring works will be considered. Because neoclassical guitar isn't very popular in de academic field of CAC, a solid lead was hard to find. In the end, solution creating algorithms for the specified genre had to be developed by the author himself. However, the literature does mention some interesting techniques. The inspiring methods from the literature used to generate music in this thesis are stochastic Markov processes, and metaheuristics such as Local Search and Ant Colony Optimization.

Experiments in this body of work are implemented in the Java programming language. The first experiments are conceptually simple and give mediocre sounding results. Nonetheless are they useful, because they spawn a learning process that ultimately leads to more sophisticated algorithms and better sounding results.

In the end, two algorithms are found that generate good sounding guitar solos. Both algorithms use a combination of Markov chains and heuristics. One algorithm uses Local Search and generates neighbours using Markov chains. The other algorithm uses an Ant Colony based algorithm, which integrates Markov transition probabilities in the algorithm itself. Both algorithms use one reference. This reference is a complete neoclassical guitar solo that provides the algorithms with a bunch of statistics. These statistics are used to build an objective function, which in its own turn gives the algorithms something to minimize. The reference is also used to estimate transition probabilities for the Markov chains. The algorithms generate solos that are melodic variations of the reference solo.

Voorwoord

Deze thesis is veruit de grootste inspanning die ik heb moeten leveren in het kader van mijn opleiding tot handelsingenieur. Naast de grote hoeveelheid aan te leren vaardigheden, zoals leren programmeren in Java en het praktisch leren toepassen van metaheuristieken, was het doorgronden van de complexiteit van het probleem een grote uitdaging. Neoklassieke gitaarmuziek als een economisch probleem definiëren was een oefening van jewelste, eens te meer omdat muziek an sich geen economisch probleem is. Het grootste probleem in deze was het probleem *zien*. Het was noodzakelijk om zelf een denkkader te ontwikkelen om zo zicht te krijgen op de zaak. Enerzijds om er vat op te krijgen, anderzijds omdat het werk in deze thesis weinig platgetreden paden volgt.

Kort gezegd, de vaardigheid die bij mij het meest is aangescherpt door het maken van deze thesis, is die van het *probleemoplossend denken*.

Ik wil dit voorwoord aangrijpen om mijn promotor, Dorien Herremans, uitvoerig te bedanken. Haar ervaring op het gebied van Computer Aided Composing heeft mij enorm vooruit geholpen, des te meer omdat er niet zo bijzonder veel mensen zijn die deze ervaring kunnen delen. De brainstorm sessies die we gehad hebben waren een belangrijke driver voor het succes van dit project.

Verder wil ik alle mensen bedanken die interesse hebben getoond in deze thesis, en mij op die manier hebben gemotiveerd om er het beste van te maken.

Max Alexander Hein

10 augustus 2014, Antwerpen

Lijst van figuren en tabellen

Inhoudsopgave

1 Onderwerp, methodiek en onderzoeksvragen

1.1 Een probleemschets

Deze thesis bevindt zich in het domein van Computer Aided Composing, of kortweg CAC. Dit houdt het genereren van muziek met behulp van algoritmes en heuristieken in. Het doel van deze thesis is om *zelf* algoritmes te ontwikkelen die muziek genereren binnen een bepaalde stijl. Daartoe moet om te beginnen onderzocht worden welke technieken er zoal gebruikt worden binnen CAC. Er moet tevens een benaderingsstrategie ontwikkeld worden, om een aanknopingspunt te vinden en om de zaak aan te kunnen pakken. Vervolgens moeten deze technieken aangewend worden om zelf algoritmes op te stellen en muziek te genereren. Via het leerproces dat onderweg optreedt, kan er nagegaan worden welke technieken en benaderingswijzen goed werken en welke minder goed. Ten slotte wordt er een conclusie geformuleerd uit de ervaringen die zich hebben voorgedaan, over welke methodes goed werken voor het genereren van muziek en hoe er op verder kan gegaan worden.

In de eerste plaats moet er een muziekstijl gekozen worden om op te focussen. Er wordt geopteerd voor neoklassieke gitaarmuziek. *Neoclassical* is in deze context een speelstijl voor de elektrische gitaar, gebaseerd op barokke en romantische klassieke muziek en gericht op een hoge mate van virtuositeit. In feite is het een verbastering van muziek van grote namen als Bach en Vivaldi, en herwerkt zodanig dat het geschikt is om te spelen door een sologitarist. Deze speelstijl vindt zijn oorsprong in de rockmuziek van de jaren 1970 en is sinds de jaren '80 een erg dominant element binnen verschillende metal genres.

Er moet onderzocht worden welke technieken gebruikt worden op het vlak van Computer Aided Composing. Aangezien het een sterk uiteenlopend domein is, wordt er afgegaan op voorbeelden die een connectie hebben met het gekozen genre of die eenvoudigweg inspirerend zijn. De focus ligt op technieken die ook hun toepassing hebben in de economische wetenschappen, gezien het feit dat deze thesis dient tot het bekomen van de graad master in de economische wetenschappen - handelsingenieur. De scope van dit werk ligt dan ook grotendeels op optimalisatieproblemen en -technieken. Om dergelijke problemen op te lossen, is het in eerste instantie nodig om een probleem te definiëren. Bijgevolg wordt muziek in wat volgt beschouwd als een probleem en tonale muziekstukken als oplossingen.

Om muziek als een probleem te kunnen zien en vervolgens op te lossen, moet er een benaderingsstrategie gekozen worden om het te beschouwen, af te lijnen en behandelbaar te maken. Dit wordt gedaan door het probleem te bekijken vanuit het perspectief van een gitarist. Dit creëert een specifiek denkkader, waardoor de grote hoeveelheid aspecten van het probleem gefilterd kunnen worden. Het gekozen perspectief is determinerend voor de oplossing, omdat bepaalde aspecten sterk benadrukt worden, terwijl andere aspecten wegvallen.

Vervolgens kan er een leerproces beginnen, waarbij de concrete toepassing van CAC geïmplementeerd wordt in Java. Aan de hand van de experimenten die hieruit voortvloeien, kan er bekeken worden welke technieken goed werken binnen de specifieke context van dit werk. Er wordt steeds verder gezocht naar betere manieren die goede resultaten leveren, vertrekkende van de tot

dan opgedane ervaringen. Hierdoor ontstaat er een evolutie van zelf opgestelde algoritmes die steeds beter aanhoorbare muziek genereren.

Uiteindelijk leiden deze ervaringen tot een conclusie, die aanreikt welke de efficiënte en effectieve methoden zijn voor het algoritmisch genereren van neoklassieke gitaarmuziek. Tot slot wordt er aangereikt hoe de auteur de volgende stappen in het onderzoek zou aanpakken.

1.2 Onderzoeksvragen

Contemplatie over het probleem leidde met vallen en opstaan tot de globale strategie uit punt 1.1. Deze strategie kan samengevat worden in een aantal vragen, vertrekkende vanuit een hoofdvraag. Centraal staat de vraag:

Wat is de beste benaderingswijze en welke zijn de meest efficiënte en effectieve heuristieken voor het algoritmisch genereren van neoklassieke gitaarmuziek?

Deze hoofdvraag kan opgesplitst worden in een aantal werkbare deelvragen. Deze zijn:

Wat zijn de mogelijke technieken en hoe wordt dit toegepast binnen Computer Aided Composing?

- → Definities en verklaringen.
- → Gekende toepassingen op CAC.

Wat is de beste benaderingswijze?

- → Muziektheoretische aanpak versus puur stochastische aanpak.
- → Bottom up (verschillende stijlelementen apart benaderen en vervolgens integreren) versus top down (het geheel ineens benaderen en eventueel stijlelementen uitdiepen).

Welke zijn de meest efficiënte en effectieve heuristieken?

- → Afweging van verschillende heuristieken.
- → Met en zonder implementatie van Markov transitiematrices.

Wat zijn de maatstaven voor efficiëntie en effectiviteit?

- → Efficiëntie: een zo snel mogelijk algoritme, het minimum aan code en inspanningen vinden om tot een goed resultaat te komen.
- → Effectiviteit: subjectief (hoe klinkt het?) versus objectief (kleinste waarde van de doelfunctie).

2 Wat is neoklassieke gitaarmuziek?

2.1 Ontstaansgeschiedenis

Bij wijze van inleiding is het nuttig om eerst te verklaren wat neoklassieke gitaarmuziek is. Wanneer mensen het woord *neoclassical* horen of lezen denken ze misschien aan neoclassicisme, een kunststijl die in het interbellum floreerde. Dit is niet juist. In deze context is neoclassical een speelstijl voor de elektrische gitaar die vooral terug te vinden is in diverse rock- en metalgenres.

De speelstijl is ontstaan rond het jaar 1970 in de progressive- en hardrockscène. De Britse rockband Deep Purple wordt doorgaans geaccrediteerd voor het uitvinden van deze neoklassieke stijl. Op zoek naar een nieuw soort rockmuziek, lieten keyboardspeler Jon Lord en gitarist Ritchie Blackmore klassieke elementen binnensijpelen in hun hardrocksound. In die tijd was er nog een grote evolutie in elektrische gitaren en gitaarversterkers, waardoor het mogelijk was om nieuwe technieken te creëren die toelieten om piano- en vioolstukken te spelen op gitaar. Het resultaat was een explosieve, snelle en technische mix tussen klassieke muziek en hardrock. Lord en Blackmore exploiteerden deze nieuwe mogelijkheden om snelle en donkere instrumentaties te maken voor Deep Purple. De muziek die daaruit voortkwam inspireerde tal van bands die een zwaardere richting uitgingen (Wikipedia, 2014).

De technische kant van neoklassieke gitaarmuziek trok een nieuw soort gitarist aan, de zogenaamde *shredders*. Deze stroming van gitaristen maakten er een sport van om zo technisch en zo snel mogelijk te spelen. Een mix van klassieke muziek en heavy metal was ideaal om dit doel te bereiken. In beide genres is snelheid geen beperking om thematisch sterke muziek te maken. Deze shredderbeweging, alhoewel nooit meer verdwenen binnen het wereldje van gitaarfreaks, kende een hausse in de jaren 1980. De meest notabele neoklassieke shredder is ongetwijfeld de zweed Yngwie Malmsteen. Onder het adagium "How can less be more? More is more." verbasterde deze gitaarheld de meest snelle en vurige klassieke muziek tot *speedmetal*. Onder zijn favoriete bronnen bevonden zich in de eerste plaats Paganini en Vivaldi (Monk, 2012).

Moderne supergitaristen hebben doorgaans een ruimer speelveld dan enkel de neoklassieke stijl, maar integreren deze stijl binnen het geheel van stijlen die ze beheersen. Bekende voorbeelden hiervan zijn Steve Vai (o.a. Frank Zappa, soloartiest) en John Petrucci (Dream Theater, soloartiest). Bijkomend zit neoklassieke gitaarmuziek verweven in het DNA van heel wat moderne metalgenres. Dit is enerzijds te wijten aan de historische invloed van bands zoals Deep Purple, anderzijds wegens de drang van veel gitaristen om snel en technisch te spelen (Marano, 2013).

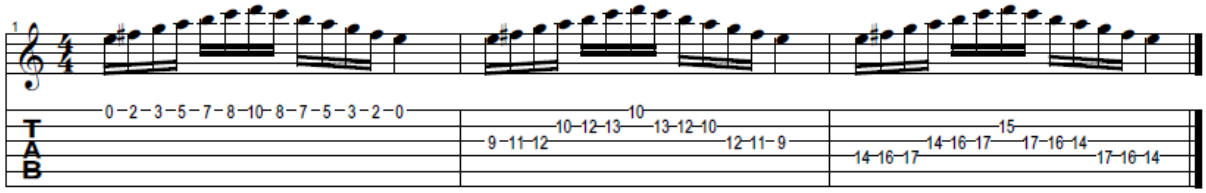
2.2 Stijlkenmerken

In de voorgaande paragraaf werd een beeld geschetst van wat neoklassieke gitaarmuziek is. Wat nog ontbreekt zijn de concrete kenmerken die neoclassical zo klassiek doen klinken. Om te beginnen zijn er de toonladders. Neoclassical wordt gedomineerd door mineurtoonladders (Ultimate Guitar, 2007). Er zijn er drie: de natuurlijke, de harmonische en de melodische mineurtoonladder. Deze laatste komt binnen de stijl niet erg vaak voor, het zijn vooral de eerste twee die voorkomen. Een volledige en correcte definitie van wat mineurtoonladders zijn wordt achterwege gelaten, want dit vereist de nodige kennis inzake muziektheorie hetgeen buiten de scope van dit werk valt. Toonladders zijn kort gezegd een vast patroon van opeenvolgende noten, vertrekkende van een basisnoot. Voor de

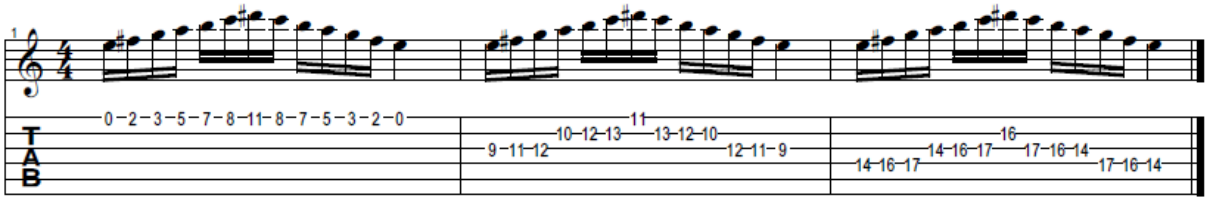
betreffende toonladders ziet dit patroon, met als eenheid intervallen van halve tonen, er noot per noot als volgt uit (Pilhofer en Day, 2009, p. 132-139):

→ Natuurlijke mineurtoonladder: 2-1-2-2-1-2-(2)
→ Harmonische mineurtoonladder: 2-1-2-2-1-3-(1)
→ Melodische mineurtoonladder: 2-1-2-2-2-1-(2) (enkel stijgend)

In dit werk wordt er vooral gekeken naar de melodische intervallen van muziek. Het gaat dan over de intervallen tussen opeenvolgende noten. Dit staat in contrast met harmonische intervallen, hetgeen duidt op de intervallen tussen noten die gelijktijdig worden gespeeld (Pilhofer en Day, 2009, p. 91-95). Kortom, toonladders identificeren was in dit werk bij momenten belangrijk en vormen in bepaalde delen een restrictie op de oplossing.

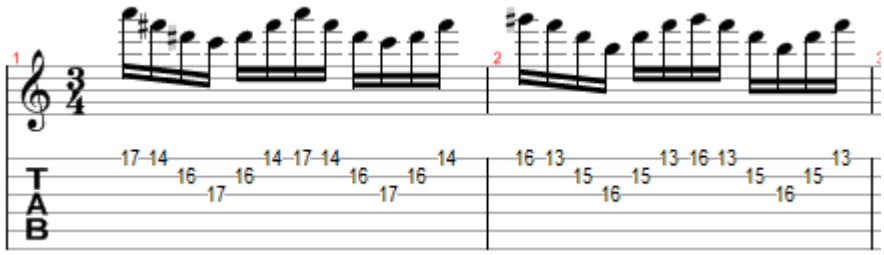


Figuur 1: de natuurlijke mineurtoonladder in E_4, gespeeld op drie manieren op de gitaarhals (van de auteur).

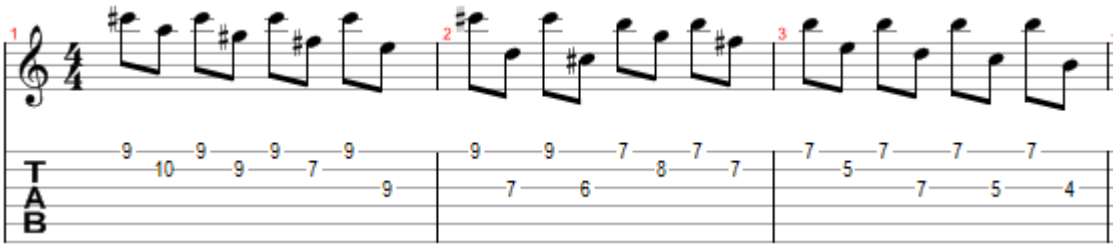


Figuur 2: de harmonische mineurtoonladder in E_4, gespeeld op drie manieren op de gitaarhals (van de auteur).

Naast de toonladders zijn er andere opvallende stijlkenmerken van neoklassieke gitaarmuziek. Twee zeer opvallende zijn arpeggio's en orgelpunten (Ultimate Guitar, 2007). Arpeggio's zijn gebroken akkoorden. Dit betekent dat een akkoord niet als geheel, maar de noten ervan sequentieel in een patroon worden gespeeld (Wikipedia, 2014). Een orgelpunt is een noot of een groep van noten die regelmatig herhaald wordt. Het orgelpunt wordt gekoppeld aan een melodie en blijft een beetje op de achtergrond. Orgelpunten worden vaak gebruikt om een melodie te versterken (Wikipedia, 2013). Omdat deze twee stijlkenmerken zo dominant zijn, zullen we steevast nakijken of ze ook in de gegenereerde oplossingen voorkomen. Op deze manier kan de effectiviteit van een algoritme concreter afgetoetst worden. Verder worden stukken binnen het geheel ook vaak herhaald, vaak in variaties ervan. Herhalingen zijn binnen neoklassieke gitaarmuziek cruciaal om een stuk thematisch sterker te maken (Ultimate Guitar, 2007).



Figuur 3: een aantal arpeggio's (uit *Arpeggio's From Hell - Yngwie Malmsteen*).



Figuur 4: een melodie gekoppeld aan een orgelpunt hoger dan de melodie zelf (van de auteur).

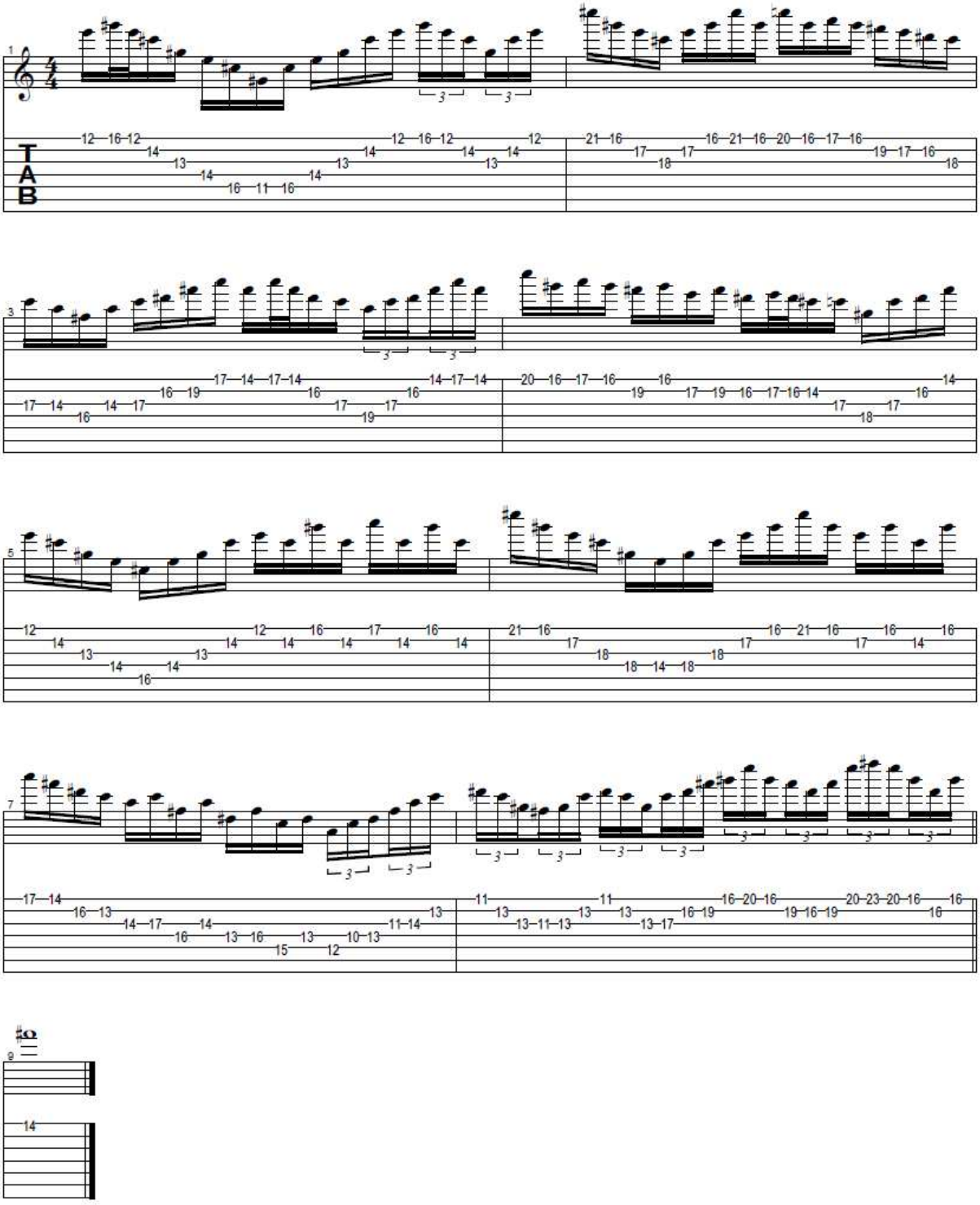
2.3 Waarom neoklassieke gitaarmuziek?

Om een muziekalgoritme te maken moet er eerst een genre gekozen worden om op te focussen. De voorkeur ging direct uit naar moderne gitaarmuziek. Enerzijds omdat dit soort muziek relatief weinig gekozen wordt binnen het kader van Computer Aided Composing, alwaar dus een nieuwe inbreng geleverd kan worden. Anderzijds omdat de auteur, amateur-gitarist zijnde, er ervaring mee heeft. Waarom dan kiezen voor *neoklassieke* gitaarmuziek? Neoclassical is een afgeleide van klassieke muziek en is dus eveneens gebonden aan tal van rigide stijlkenmerken, in tegenstelling tot bijvoorbeeld jazz fusion. Bovendien is het weinig expressief. Hiermee wordt bedoeld dat het een eerder technische stijl is waar er gefocust wordt op het spelen van veel noten, eerder dan veel uit te drukken met weinig noten. Dit deed bij de auteur een vermoeden rijzen dat een methodische, en dus mogelijk algoritmische, manier van componeren binnen de stijl mogelijk is. Bovendien kan binnen deze stijl de opeenvolging van noten als discreet beschouwd worden. Andere stijlen, zoals bijvoorbeeld blues rock, maken gebruik van niet weg te denken speeltechnieken waarvoor de overgang tussen noten enkel als continu beschouwd kan worden (zoals het vaak voorkomen van *slides* en vooral *bends* van noten). Muziek bekijken als een discrete opeenvolging van noten faciliteert de implementatie in een computeromgeving. Bijkomend betreft het in veel gevallen quasi monofone muziek, zodat harmonische relaties binnen de stijl verwaarloosbaar zijn. Ritme is ook niet determinerend voor het genre, in tegenstelling tot ska bijvoorbeeld. Kortom, van alle moderne gitaarstijlen leek neoclassical de meest toegankelijke om te verwerken in een algoritme.

2.4 Referentiesolo

Figuur 5 geeft een volledige neoklassieke gitaarsolo weer. Deze is afkomstig van de Amerikaanse progressive metalband Dream Theater. Deze solo klinkt erg barokachtig en omvat kort maar krachtig zowat alle kenmerken van de neoklassieke gitaarstijl. Zo zijn er verschillende arpeggio's in terug te vinden, eenvoudig te onderscheiden aan de hand van de sterke golfbewegingen op de partituur. Hier en daar is er een orgelpunt te horen en de onmiskenbare neoklassieke stijl ervan duidt op het gebruik van mineurtoonladders. Tevens wordt hetzelfde thema in variaties enkele malen herhaald.

Deze solo zal in dit werk vaak dienen als referentiesolo. Enerzijds als input voor algoritmes, anderzijds als 'de ideale neoklassieke gitaarsolo' waartegenover de output van experimenten subjectief vergeleken zou kunnen worden.



Figuur 5: een volledige neoklassieke gitaarsolo (uit *Build Me Up, Break Me Down - Dream Theater*).

3 Algoritmische muziek en Computer Aided Composing

Muziek is een complex gegeven. Mensen associëren muziek met emotie, maar er zit ook een groot aandeel logica in. Of het nu gaat over de fysische eigenschappen van instrumenten en de golven die ze voortbrengen of de theorie van het componeren, muziek is altijd logisch of zelfs mathematisch uit te drukken. Een zondagsfilosoof zou zelfs kunnen stellen dat muziek de enige creatie van de mens is waar logica en emotie harmonieus samengaan.

Naast het logisch beschrijven van muziek, zijn componisten ook altijd geïntrigeerd geweest door het logisch componeren van muziek. Een bekend historisch voorbeeld hiervan is het dobbelsteenspel van Mozart. In dit spel worden twee dobbelstenen gegooid. Elke geworpen som wordt gekoppeld aan een kort stukje vooraf geschreven muziek. Door telkens opnieuw met de dobbelstenen te gooien vormt er zich een muziekstuk dat op probabilistische wijze is gegenereerd. Aangezien de kansen op de sommen van 2 dobbelstenen niet uniform verdeeld zijn, zullen sommige stukjes meer herhaald worden dan andere. Door de stukjes muziek stuk per stuk strategisch aan een bepaalde som van twee dobbelstenen te koppelen, kan er toch zinvol klinkende muziek gevormd worden. Op deze manier creëerde Mozart een mooi voorbeeld van algoritmisch componeren (Mosteller en Youtz, 1968).

De vraag is of het componeren van muziek niet inherent heuristisch is. Muziek maken is zelden een kwestie van zuivere genialiteit, maar eerder een incrementeel proces van leren, ervaring opdoen, invloeden laten spreken, veel muziek produceren, het selecteren van de beste stukken uit alle gegenereerde stukken en zwoegen om iets te doen werken. De methodische manier waarop een muziekartiest te werk gaat heeft veel weg van een computeralgoritme (Jacob, 1996). De eerste experimenten in dit werk zijn gebaseerd op de heuristische manier waarop een onervaren muzikant muziek zou maken, vertrekkende van een nauwe kennis en geen ervaring. Achteraf bleek dit weliswaar niet de meest efficiënte methode te zijn.

Door het ontstaan van de computertechnologie, zijn de mogelijkheden voor algoritmisch componeren drastisch uitgebreid en bovendien een stuk gesofisticeerder geworden dan het spel van Mozart. Het eerste experiment in het domein van Computer Aided Composing werd uitgevoerd door de Amerikaan Lejaren Hiller op een Illiac computer, een van de eerste computers gebouwd na de tweede wereldoorlog. In 1956 resulteerde zijn werk in het eerste computergegenereerde muziekstuk genaamd *The Illiac Suite For String Quartet*. Hiller maakte gebruik van Markov ketens en random walk algoritmes om zijn muziek te genereren (Edwards, 2011).

In de tijd van Hiller waren computers unieke machines met weinig rekenkracht. Het domein van CAC is dus pas echt kunnen groeien met het tot wasdom komen van de computertechnologie. In het heden, waar computers overal beschikbaar zijn en bovendien over voldoende rekenkracht beschikken om experimenten in het gebied van CAC uit te voeren, is het aantal van deze experimenten dan ook geëxplodeerd (Assayag, 1998).

Men zou een boek vol kunnen schrijven over de ontwikkelingen die hebben plaatsgevonden sinds Hiller, hetgeen hier achterwege wordt gelaten wegens weinig relevant. Elk genre of stijl van muziek vereist een eigen aanpak, en pogingen om muziek algoritmisch te genereren kunnen hoe dan ook sterk uiteenlopen. Onderzoek in CAC vormt dus allesbehalve een coherent geheel. Aangezien neoklassieke gitaarmuziek nog min of meer onontgonnen terrein blijkt te zijn in de wereld van CAC, kan er ook geen echte coherente overgang gemaakt worden van reeds bestaand onderzoek naar dit

werk. Desalniettemin bestaan er wel enkele inspirerende bronnen die hebben geholpen dit werk te verwezenlijken en deze worden in wat volgt kort uit de doeken gedaan.

3.1 Contrapuntmuziek met Variable Neighbourhood Search

In hun paper *Composing Fifth Species Counterpoint Music With Variable Neighbourhood Search* beschrijven Dorien Herremans en Kenneth Sörensen (2012) hoe ze contrapuntmuziek genereren door een combinatie te maken van compositorische regels en optimalisatietechnieken.

Het componeren van contrapuntmuziek is in principe gebonden aan een stel concrete regels, opgesteld door ene Johann Fux in 1725. Contrapuntmuziek zou zo goed mogelijk aan deze regels moeten beantwoorden. Met andere woorden, in een CAC omgeving moet de afwijking van algoritmisch gegenereerde muziek ten opzichte van deze regels geminimaliseerd worden en bijgevolg wordt muziek bekeken als een optimalisatieprobleem. Er wordt een doelfunctie opgesteld die bestaat uit een som van gewogen scores. Deze scores geven de afwijking weer van een gecreëerde oplossing ten opzichte van de individuele contrapuntregels van Fux. De waarde van de doelfunctie, de som van de scores van de gevonden oplossing, moet dan zo klein mogelijk zijn. Om een goede oplossing te genereren wordt er gebruik gemaakt van een Variable Neighbourhood Search algoritme. Hierbij wordt er eerst een willekeurig stuk muziek gegenereerd. Vervolgens worden er kleine aanpassingen gedaan in dit stuk volgens in het algoritme vastgelegde principes. Dan wordt er nagegaan welke van deze aanpassingen de beste fit heeft met de regels van Fux. De beste aanpassing wordt steeds overgehouden. Over een groot aantal iteraties vormt er zich dan een goede oplossing, zijnde een stuk contrapuntmuziek dat nauw aansluit bij de compositorische regels van Fux (Herremans en Sörensen, 2012).

In dit werk zal muziek ook benaderd worden als een optimalisatieprobleem. Er wordt dieper ingegaan op optimalisatieproblemen en technieken om deze op te lossen, zoals VNS, in deel 4. Het aandeel muziektheorie in de experimenten zal echter minimaal zijn of zelfs volledig achterwege gelaten worden, omdat er voor neoklassieke gitaarmuziek geen hapklare regels bestaan die aangeven hoe een compositie gemaakt moet worden.

3.2 Melodieën maken met behulp van Markov ketens

François Pachet en Pierre Roy (2010) beschrijven in hun werk *Markov constraints: steerable generation of Markov sequences* hoe ze met behulp van Markov ketens solo's kunnen genereren in de stijl van de virtuoze jazzgitarist Al Di Meola. De gitarist staat bekend om zijn razendsnelle mix tussen jazz, rock en mediterrane invloeden. Hij maakte onder andere furore met de band Return To Forever en het gitaartrio De Lucia, McLaughlin & Di Meola.

Een Markov keten wordt gemaakt door na te gaan wat de kansen zijn dat een element in de keten gevolgd wordt door een van de andere mogelijke elementen. De opvolging van element tot element gebeurt dus probabilistisch. Deze kansen, genaamd transitiekansen, moeten geschat worden. Hier bestaan verschillende methoden voor, van zeer eenvoudig tot zeer complex. Markov ketens worden in meer detail besproken in deel 5. Pachet en Roy maakten Markov ketens voor solomelodieën aan de hand van exotische toonladders die Al Di Meola vaak gebruikt. Op deze manier konden ze solo's in zijn kenmerkende stijl genereren (Pachet en Roy, 2010).

Net zoals de neoklassieke shredders, is de stijl van Al Di Meola zeer technisch en eerder weinig expressief. Aangezien bovenstaand vermeld experiment aantoont dat het mogelijk is om korte,

technische solo's te genereren met behulp van Markov ketens, komen deze ook regelmatig terug in dit werk.

3.3 Melodieën maken met behulp van Ant Colony Optimization

Een klein aantal auteurs is er in geslaagd om met behulp van Ant Colony Optimization muziek te genereren. ACO is een metaheuristiek gebaseerd op het groepsgedrag in mierenkolonies. Meer over ACO is terug te vinden in deel 4.1.2. Er is een experiment terug te vinden waarbij men een ACO gebaseerd algoritme heeft gebruikt om jazz solo's te genereren (Bäckman, 2010) en een ander voorbeeld waarbij men een dergelijk algoritme heeft gebruikt om barokke melodieën te maken (Geis en Middendorf, 2008).

In een eerder werk heeft de auteur uitvoerig het ontstaan en de werking van Ant Colony Optimization besproken (De Wieuw, Gijsbrechts, & Hein, 2012). Wegens persoonlijke interesse komt ACO ook in dit werk terug als basis om een algoritme te ontwikkelen in deel 7.3.2.

4 Optimalisatieproblemen

Veel realistische problemen kennen niet één, maar meerdere oplossingen. Bijvoorbeeld, als men van Antwerpen naar Brussel reist zijn er verschillende routes en vervoersmiddelen af te wegen. Stel dat men een muziekstuk wil maken bestaande uit noten van een en dezelfde toonladder, dan is het aantal 'muziekstukken' dat men met deze noten kan maken schier eindeloos. Dergelijke problemen geven aanleiding tot optimalisatie; men wil uit de mogelijke oplossingen de beste oplossing selecteren of toch minstens een van de betere. Men wil de goedkoopste trip van Antwerpen naar Brussel plannen en men wil een aanhoorbaar muziekstuk maken met noten uit de natuurlijke mineurschaal in E.

Formeler bestaat een optimalisatieprobleem uit 3 delen (Parys en Pauwels, 2010, p. 117-120):

- → een doelfunctie
- → beslissingsvariabelen
- → restricties.

De doelfunctie of economische functie staat centraal in het optimalisatieprobleem en is de functie die geminimaliseerd of gemaximaliseerd moet worden. De doelfunctie wordt beschreven in functie van de beslissingsvariabelen. De beslissingsvariabelen zijn eenvoudigweg de veranderlijken waar een keuze over kan gemaakt worden en die een invloed hebben op de kwaliteit van de oplossing. Restricties zijn een bijkomend aspect die beperkingen doorvoeren op de beslissingsvariabelen en uiteindelijk ook indirect op de doelfunctie zelf. Met behulp van restricties kan er een oplossingsruimte afgebakend worden (Springael, 2013,p. 2-6).

4.1 Oplossingsmethoden voor optimalisatieproblemen

De oplossingsmethoden die in dit werk gebruikt worden zijn *metaheuristieken*. Het woord metaheuristiek komt van de Griekse woorden *meta* en *heuriskein* en betekent zoveel als high level zoekstrategie (Sorënsen en Glover, n.d.):

"Een metaheuristiek is een high level, probleemonafhankelijk, algoritmisch denkkader dat een strategie aanbiedt om een zoekalgoritme voor optimalisatieproblemen te ontwikkelen."

De metaheuristieken die hier besproken zullen worden zijn *Local Search*, *Variable Neighbourhood Search* en *Ant Colony Optimization*.

4.1.1 Local Search en Variable Neighbourhood Search

Local Search algoritmes zijn algoritmes die iteratief van de ene oplossing naar een steeds betere oplossing gaan. Ze bestaan doorgaans uit 4 stappen (Pinedo, n.d.):

- → Initialisatie: stel een eerste aanvaardbare oplossing *s* op en bereken daarvan de doelfunctiewaarde *F(s).*
- → Maak een of meerdere buren: een buur *s'* is een variatie op *s* aan de hand van een vooraf bepaald principe. Bereken van deze buur *s'* tevens de doelfunctiewaarde *F(s').*
- → Acceptatietest: test of de oplossing *s'* al dan niet beter is dan *s*. Maak in functie daarvan de keuze om *s'* al dan niet te aanvaarden als de nieuwe oplossing.
- → Afsluitingstest: bepaal aan de hand van zelf opgestelde criteria of het algoritme al dan niet moet stoppen. Zo niet, ga dan terug buren maken.

De eerste oplossing is doorgaans een volledige en willekeurig gegenereerde oplossing die aan de restricties voldoet. De buren van deze oplossing kunnen volgens verschillende principes gemaakt worden. In dit werk worden er vaak een of meerdere opeenvolgende elementen in de oplossing vervangen door andere elementen van buiten de oplossing. Er wordt dan altijd op voorhand vastgelegd welke 'bouwstenen', bijvoorbeeld de noten van een toonladder, gekozen kunnen worden. Men spreekt van *Variable Neighbourhood Search* in plaats van Local Search indien meerdere methodes om buren te genereren elkaar afwisselen (Sörensen en Glover, n.d.). Dit gebeurt dan weer aan de hand van regels die de ontwerper van het algoritme vaststelt. Indien bijvoorbeeld de ene methode om buren te genereren geen betere oplossing oplevert en het algoritme vastzit in een lokaal optimum, dan wordt er overgegaan naar een andere methode om buren te genereren. Deze manier van zoeken levert vaak betere resultaten op dan het gebruik van een enkele methode om buren te genereren (Mladenovic en Hansen, 1997). Sensu stricto wordt er in dit werk echter geen gebruik gemaakt van VNS.

De acceptatietest kan tevens op verschillende manieren gebeuren. Men zou kunnen kiezen voor een *steepest descent/ascent* strategie, of voor een *first improving* strategie. Bij de steepest descent/ascent strategie worden er meerdere buren gecreëerd en wordt de beste oplossing daaruit gekozen. Bij first improving wordt de eerste buur gekozen die beter is dan de huidige oplossing (Sörensen en Glover,n.d.). In dit werk wordt er gebruik gemaakt van beide strategieën. Bijvoorbeeld, in het gegenereerde muziekstuk wordt een bepaald element vervangen om buren te creëren. Een gehanteerde steepest descent strategie is dan om deze noot te vervangen door alle mogelijke noten die een toelaatbare oplossing opleveren. Uit de groep van nieuwe oplossingen (buren) wordt dan de beste gekozen, indien deze een lagere doelfunctiewaarde heeft dan de huidige oplossing. Een first improving strategie kan zijn om de noot willekeurig te blijven veranderen tot deze een betere oplossing oplevert dan de huidige oplossing.

De afsluiting van het algoritme kan gebeuren aan de hand van het aantal opgelegde iteraties, of wanneer de run time te groot wordt (Pinedo, n.d.). De eerste experimenten in dit werk stoppen na een aantal opgelegde iteraties. Andere algoritmes hebben een dynamisch aantal iteraties. Dit wordt verder besproken in deel 7.

4.1.2 Ant Colony Optimization

Wegens persoonlijke interesse van de auteur (De Wieuw, Gijsbrechts en Hein, 2012) is er in dit werk geëxperimenteerd met op Ant Colony Optimization gebaseerde algoritmes, hetgeen terug te vinden is in deel 7.3.2.

ACO is een metaheuristiek gebaseerd op het groepsgedrag van mieren. Als mieren op zoek gaan naar voedsel zwermen ze uit vertrekkende van hun nest. Wanneer een mier voedsel vindt, dan laat het op de terugweg naar het nest een feromonenspoor achter, zodat andere mieren de weg naar de voedselbron kunnen traceren. Andere mieren zullen het pad van deze mier beginnen volgen wegens de aantrekking van de feromonen. Hoe sterker de concentratie van de feromonen op een spoor, des te harder zal een mier zich aangetrokken voelen om dat spoor te volgen. Over de tijd verdampen deze feromonensporen echter. Hierdoor zullen mieren een korte afstand tot de voedselbron prefereren boven een lange afstand, aangezien een mier er langer over doet om de lange afstand af te leggen en het feromonenspoor op dit pad daardoor sterker verdampt zal zijn. Door dit effect zullen de kortere paden steeds meer mieren aantrekken en zullen lange paden op termijn verlaten

worden. Uiteindelijk zullen alle mieren convergeren naar het kortste pad en ontstaat er een colonne van mieren die tussen nest en voedsel foerageren (Dorigo en Stützle, 2004, p. 4-5).

Ant System

De principes van het gedrag van mierenkolonies werden verwerkt tot een groep metaheuristieken genaamd Ant colony Optimization. Ant System was de eerste van die soort en is de basis voor andere ACO metaheuristieken. Ant System vormt tevens de basis van het algoritme in deel 7.3.2. In wat volgt wordt het in detail uit de doeken gedaan (Dréo, Pétrowski, Siarry en Taillard, 2006, p. 130-131).

Tijdens elke iteratie $t < t_{max}$ zal elke mier k $(k=1,\ldots,m)$ een volledige oplossing maken van n elementen. Op een gegeven moment bevindt mier k zich in punt i en wil het naar het volgende punt j reizen. Dit zal afhankelijk zijn van:

- → De lijst van mogelijke punten J_i^k die mier k kan bezoeken vanuit punt i.
- → De zogenaamde *zichtbaarheid* η_{ij} van j ten opzichte van i, invers gerelateerd aan de afstand d_{ij} tussen de twee punten: $\eta_{ij} = 1/d_{ij}$.
- → De *intensiteit* $\tau_{ij}(t)$ van het feromonenspoor op het pad (i,j) tijdens het moment of iteratie t.

De kans dat mier k van een punt i naar een punt j gaat hangt af van de *random proportional transition rule*. Deze wordt gegeven door de formule:

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \,\epsilon\, J_i^k}[\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} \; als\; j \,\epsilon\, J_i^k, anders\; 0. \tag{4.1}$$

De kans $p_{ij}^k(t)$ dat een mier k van i naar j gaat is het gewogen product van de intensiteit en de zichtbaarheid tussen i en j, gedeeld door de som van de gewogen producten van de intensiteit en de zichtbaarheid tussen i en alle mogelijke punten l vanuit i. Indien j onbereikbaar is, dan is $p_{ij}^k(t) = 0$. De gewichten α en β zijn complementaire parameters tussen 0 en 1, vast te leggen door de gebruiker van het algoritme. Is α=0, dan worden de feromonensporen uitgeschakeld. Is β=0, dan wordt de keuze van mier k uitsluitend bepaald door de feromonensporen. De keuze voor α en β is een afweging tussen intensificatie (hoge α) en diversificatie (hoge β) van de keuzemogelijkheden van mier k.

De mier moet voor elke i opnieuw een keuze voor j maken, totdat het een oplossing van n elementen heeft gemaakt. Uiteindelijk zal mier k een pad $T^k(t)$ hebben gemaakt met lengte $L^k(t)$ en laat het een feromonenspoor $\Delta\tau_{ij}^k(t)$ achter op het pad volgens de formule:

$$\Delta\tau_{ij}^k(t) = \frac{Q}{L_k(t)} \; als\; (i,j) \,\epsilon\, T^k(t), anders\; 0. \tag{4.2}$$

Q is een parameter, te bepalen door de gebruiker. In dit werk is Q altijd $Q=1$. Tot slot moeten oude feromonensporen geëvaporeerd en geüpdatet worden. Dit gebeurt aan de hand van de formule:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{4.3}$$

De parameter ρ staat voor de evaporatiecoëfficiënt en wordt vastgelegd door de gebruiker. Het is evident dat hoe hoger ρ is, hoe sneller de sporen geëvaporeerd worden.

Ter overzicht wordt Ant System nog eens doorlopen. Per iteratie *t* en vervolgens per mier *k*, wordt er een oplossing gebouwd van *n* elementen. Deze elementen worden element per element, van *i* naar *j* gekozen aan de hand van de random proportional rule. Deze regel staat in functie van de afstand tussen de twee punten enerzijds, en het feromonenspoor tussen de twee punten anderzijds. Wanneer mier *k* een oplossing gebouwd heeft, laat het een feromonenspoor achter op de paden die het gepasseerd heeft. Tot slot worden op het einde van elke iteratie alle aanwezige sporen voor een deel geëvaporeerd. Uiteindelijk zouden na een groot aantal iteraties alle mieren moeten convergeren naar een goede oplossing voor het vooropgestelde optimalisatieprobleem (Dorigo en Stützle, 2004, p. 70-73). Hoe AS concreet wordt toegepast voor het genereren van neoklassieke gitaarmuziek is te lezen in deel 7.3.2.

4.1.3 Gerelateerde begrippen

In de volgende delen zullen er nog enkele begrippen aan bod komen die betrekking hebben op de oplossingsmethoden uit voorgaande paragrafen. Deze worden hier kort uit de doeken gedaan.

Perturbatie

Het kan gebeuren dat een Local Search algoritme vast komt te zitten in een lokaal optimum. De buren die gegenereerd worden liggen dan te dicht bij dit optimum, waardoor een nieuwe en betere oplossing nooit gevonden wordt. Om hieraan te ontsnappen kan men gebruik maken van de perturbatietechniek. Dit klinkt speciaal, maar eigenlijk wil het zeggen dat een groot aantal elementen in de oplossing willekeurig veranderd wordt. De buren die na de perturbatie gegenereerd worden, zullen dan ver genoeg van het lokale optimum liggen om er niet opnieuw in terecht te komen (Sörensen en Glover, n.d.). Perturbaties in dit werk gebeuren door 5-10% van de elementen in de oplossing willekeurig te veranderen.

Tabu Lists

Soms kan het zijn dat een oplossing regelmatig terug naar voor komt in een algoritme, bijvoorbeeld een lokaal optimum dat telkens opnieuw gekozen wordt als beste oplossing in een Local Search Algoritme. Om te vermijden dat een algoritme keer op keer in dezelfde situatie terechtkomt, maakt men gebruik van zogenaamde Tabu Lists. Men steekt oplossingen of elementen van een oplossing in een geheugenlijst, de Tabu List, waardoor men ervoor kan zorgen dat deze elementen of oplossingen niet meer in het algoritme voorkomen (Glover, 1990).

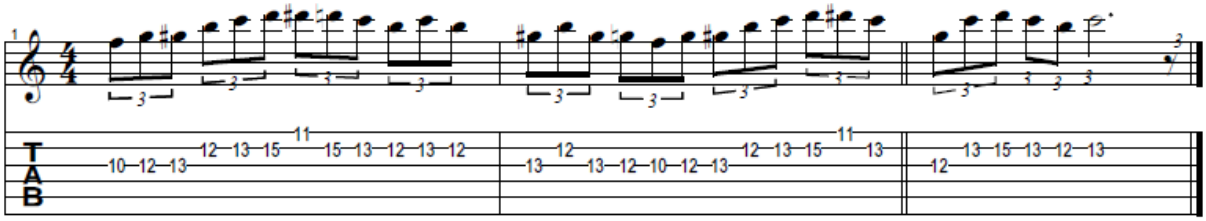
5 Markov ketens

Een Markov keten is een discreet stochastisch proces. Hierbij wordt er aan de hand van een bepaalde kansverdeling overgegaan van de ene naar de andere toestand (Anderson, 2011).

Er is een verzameling $S = \{s_1, s_2, ..., s_r\}$ bestaande uit r aantal mogelijke toestanden s. Het kansproces van de Markov keten begint in een bepaalde toestand s_i en gaat stapsgewijs over naar de volgende toestand s_j . De kans p_{ij} dat deze stap gebeurt wordt de transitiekans genoemd en is enkel afhankelijk van de huidige toestand s_i . Voor elke mogelijke toestand $s_i \in S$ kan men een kans opstellen voor de transitie naar elke mogelijke toestand $s_j \in S$, al dan niet gelijk aan 0. Door al deze kansen in een $r \times r$ matrix te steken bekomt men een transitiematrix P, die dus voor elke stap s_i naar s_j beschrijft wat de transitiekans p_{ij} is (Grinstead en Snell, 1997, p. 405-406).

$$P = \begin{bmatrix} p_{11} & \cdots & p_{1r} \\ \vdots & \ddots & \vdots \\ p_{r1} & \cdots & p_{rr} \end{bmatrix} waarbij\ 0 \leq\ p_{ij} \leq 1\ en \sum_{j=1}^{r} p_{ij} = 1. \tag{5.1}$$

Een transitiematrix kan men voorstellen zoals in formule 5.1. Merk op dat alle elementen in deze matrix tussen 0 en 1 liggen en dat de rijtotalen moeten optellen tot 1 (Anderson, 2011).



Figuur 6: een loopje in de harmonische mineurschaal (van de auteur).

In dit werk worden transitiematrices altijd opgesteld aan de hand van een referentiesolo, of zelfs meerdere inputs. Om te illustreren hoe dit in zijn werk gaat, wordt er een Markov transitiematrix opgesteld van het stukje muziek uit figuur 6. Om een kans p_{ij} te bepalen, beschouwen we de toestand s_i . Voor deze toestand s_i gaan we na hoeveel keer een bepaalde toestand s_j volgt. Vervolgens schatten we de kans p_{ij} door het aantal keer dat s_j voorkomt na s_i te delen door het totaal aantal keren dat s_i voorkomt. Formule 5.2 geeft dit weer, waarbij n_{ij} het aantal keer is dat s_{ij} voorkomt. Dit doen we voor alle r^2 combinaties van i en j om een volledige transitiematrix te bekomen (Jones, 2005).

$$p_{ij} = \frac{n_{ij}}{\sum_{j=1}^{r} n_{ij}} \tag{5.2}$$

We bekijken 1 toestand, namelijk de eerste noot. Lezers die geen notenschrift kunnen lezen focussen zich het best op de TAB notatie. We bekijken dus noot '10' op de derde lijn van boven geteld. Dit komt overeen met de noot op de tiende fret en de derde snaar van een gitaar. TAB notatie is een laagdrempelige vorm van muzieknotatie, hetgeen erg nuttig is voor aspirant muzikanten. We zien dat deze toestand twee keer voorkomt en beide keren wordt het gevolgd door de noot op fret 12 van de derde snaar. Bijgevolg is de kans dat '12' op '10' volgt op de derde snaar gelijk aan 1. Het loopje is beperkt tot een diatonische toonladder van een octaaf, waardoor er maar zeven mogelijke toestanden zijn. De transitiematrix zal dus een 7x7 matrix zijn met 49 elementen of kansen. In een project met de naam *markovchainer* wordt het maken van de transitiematrix geautomatiseerd in een

class genaamd *NotesMarkovizer*. Het Java project *markovchainer* is integraal terug te vinden in bijlage II. De volledige transitiematrix voor het stukje muziek uit figuur 6 is te zien in tabel 1. Merk op dat de noten in de tabel worden weergegeven als MIDI nummers. In MIDI krijgt elke noot een vast getal toegewezen. Noten '10' en '12' op de derde snaar komen hier overeen met nummers 65 en 67. Muziek wordt als een serie van MIDI nummers ingegeven in de gemaakte algoritmes en komen er ook als MIDI nummers terug uit. Meer informatie over MIDI is te lezen in deel 6.

	65	67	68	71	72	74	75
65	0	1	0	0	0	0	0
67	0,25	0	0,5	0	0,25	0	0
68	0	0,25	0	0,75	0	0	0
71	0	0	0,33	0	0,67	0	0
72	0	0,14	0	0,43	0	0,43	0
74	0	0	0	0	0,5	0	0,5
75	0	0	0	0	0,5	0,5	0

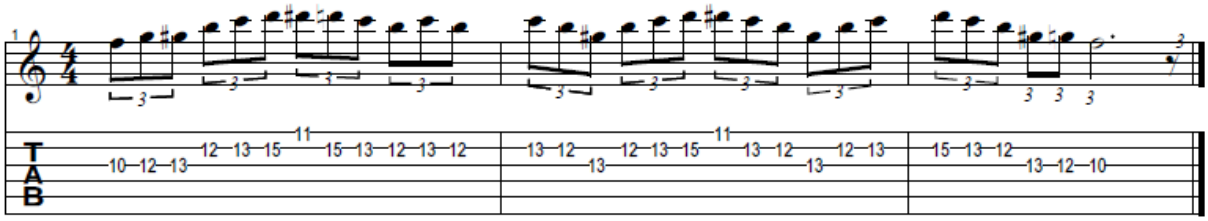
Tabel 1: transitiematrix van het loopje uit figuur 6.

Na het opstellen van de transitiematrix kan er een Markov keten gemaakt worden aan de hand van deze transitiematrix. De transitie van noot *i* naar noot *j* wordt gedaan aan de hand van de transitiekansen. Om ervoor te zorgen dat transities met een kleine kans soms ook gekozen worden, wordt de gecumuleerde transitiekans over de rij van *i* afgewogen tegen een willekeurig gegenereerde uniform verdeelde kans. Dit wordt logisch weergegeven met het stukje pseudocode in figuur 7.

```
for(int i=0;i<aantal noten;i++){

    if(i==0){
      Markovketen.add(willekeurige eerste noot);
    }
    else{
         gecummuleerde kans P=0;
          willekeurige kans random;

         for(int j=0;j<(r+1);j++){

              if(P< random){
                 P+= Transitiematrix(rij(i-1) en kolom(j));
              }
              else{
                  nieuw element= toestand(j-1);
                  break;
              }
          }
          Markovketen.add(nieuw element);
    }
}
```

Figuur 7: pseudocode voor het selecteren van de volgende noot aan de hand van de transitiekansen.

Deze code is eveneens terug te vinden in de class *MarkovChainer* van het project *markovchainer* in bijlage II.1. Met behulp van deze code kan dan een Markov keten van noten gemaakt worden, waarvan een resultaat te zien is in figuur 8.



Figuur 8: een gegenereerde Markov keten van noten, gebaseerd op het muziekstuk uit figuur 6.

Markov ketens komen op allerlei manieren terug in dit werk. Niet enkel om oplossingen te genereren, maar ook verwerkt in de doelfunctie van algoritmes.

6 MIDI, MIR en jSymbolic

Om muziek te verwerken en bewerken in algoritmes wordt er gebruik gemaakt van MIDI bestanden. MIDI staat voor *Musical Instrument Digital Interface* en is een communicatieprotocol dat interactie toelaat tussen digitale muziekinstrumenten en computers. Een MIDI bestand bestaat niet uit noten, maar uit informatie over die noten (Hansen, n.d.). Aangezien er gewerkt wordt met de melodische relaties van noot tot noot, zijn we vooral geïnteresseerd in hoe de frequenties van noten worden weergegeven in MIDI. Dit wordt gedaan door muzieknoten te koppelen aan een MIDI nummer *m*. Formule 6.1 geeft de relatie weer tussen de frequentie van een noot f_m en het overeenkomstige MIDI nummer (Wolfe, n.d.).

$$m = 12 \cdot log_2\left(\frac{f_m}{440}\right) + 69 \Leftrightarrow f_m = 440 \cdot 2^{(m-69)/12} \qquad (6.1)$$

MIDI nootnummers zijn natuurlijke getallen. Het interval tussen twee opeenvolgende MIDI nootnummers komt overeen met een halve noot. Volgens de *General MIDI Standard* moet een frequentie van 440 Hz overeenkomen met MIDI nummer 69, andere MIDI nummers worden vastgelegd aan de hand van deze referentie. Zo wordt de noot C_4 weergegeven met MIDI nummer 60 en C_4# weergegeven met MIDI nummer 61. C_4 is de centrale noot op de piano en komt overeen met een frequentie van 261,63 Hz (Wolfe, n.d.). Jammer genoeg lappen vele softwareontwikkelaars deze standaard aan hun laars en verschuiven ze MIDI nummer 60 naar eigen believen een octaaf naar boven of beneden. MIDI nummer 60 kan dan overeenkomen met C_3 of C_5 (MakeMusic, 2011). Een eenvoudig experiment leert ons dat de gebruikte muzieknotatie- en MIDI software *Guitar Pro 5* de noot C_4 koppelt aan MIDI nummer 60. Bijgevolg refereert GP5 naar MIDI nummers volgens de General MIDI conventies .

Muziekstukken worden verwerkt in de algoritmes van volgende delen als een array of vector van MIDI nootnummers. Figuur 9 geeft de referentiesolo uit figuur 5 weer als een array van MIDI nummers.

```
//Referentiesolo Dream Theater - Build Me Up, Break Me Down @4min20
int[] MidiSeq= new int []{
76,80,76,73,68,64,61,56,61,64,68,73,76,80,76,73,68,73,76,85,80,76,73,76,80,85,80,8
4,80,81,80,78,76,75,73,72,69,66,69,72,75,78,81,78,81,78,75,72,69,72,75,78,81,78,84
,80,81,80,78,80,76,78,75,76,75,73,72,68,72,75,78,76,73,68,64,61,64,68,73,76,73,80,
73,81,73,80,73,85,80,76,73,68,64,68,73,76,80,85,80,76,80,73,80,81,78,75,72,69,72,6
6,69,63,66,60,63,57,60,63,66,69,72,75,72,68,66,68,72,75,72,68,72,75,78,80,84,80,78
,75,78,84,87,84,80,75,80,78};
```

Figuur 9: voorstelling van de referentiesolo als een array van MIDI nummers.

```
int[] E4NaturalMinor = new int []{64,66,67,69,71,72,74,76};
```

Figuur 10: voorstelling van de natuurlijke mineurschaal in E_4 plus de octaafnoot als een array van MIDI nummers.

Om iets zinvol te kunnen doen met deze MIDI nummers, moeten er een hoop statistieken uit gehaald worden. Dit is het domein van *Music Information Retrieval*, of kortweg MIR. MIR houdt zich, zoals de naam impliceert, bezig met het halen van allerlei informatie uit muziek (Downie, 2003). Om informatie uit muziekstukken te halen, wordt er gebruik gemaakt van het softwarepakket *jSymbolic*.

Hiermee is het mogelijk om tal van statistieken uit MIDI files te halen met betrekking tot, onder andere, de melodische aspecten van een muziekstuk. De 21 gebruikte statistieken in dit werk zijn terug te vinden in tabel 2, samen met een beschrijving van wat ze inhouden. Allemaal hebben ze betrekking tot de melodische aspecten van een muziekstuk (McKay, 2010). Om deze statistieken gemakkelijk te kunnen gebruiken in algoritmes, heeft de auteur ze allemaal opnieuw geprogrammeerd in statische Java methods. Dit is integraal terug te vinden in bijlage III.

Naam statistiek	**Beschrijving**
Amount of Arpeggiation	Het relatieve aandeel van de intervallen met grootte 0, 3, 4, 7, 10, 11, 12, 15 of 16.
Average Melodic Interval	De grootte van het gemiddelde interval.
Chromatic Motion	Het relatieve aantal intervallen van een halve noot (of dus MIDI intervallen van 1).
Direction of Motion	Het aantal stijgende melodische bewegingen, gedeeld door het totaal aantal melodische bewegingen.
Distance Between Most Common Melodic Intervals	Het absolute verschil tussen de twee meest voorkomende intervallen.
Duration of Melodic Arcs	Het gemiddelde aantal noten tussen melodische pieken en dalen.
Interval Between Strongest Pitches	Het absolute interval tussen de twee meest voorkomende noten.
Melodic Fifths	Het relatieve aantal intervallen van grootte 5.
Melodic Octaves	Het relatieve aantal intervallen van grootte 12.
Melodic Thirds	Het relatieve aantal intervallen van grootte 3 of 4.
Melodic Tritones	Het relatieve aantal intervallen van grootte 6.
Most Common Melodic Interval	Het meest voorkomende interval.
Most Common Melodic Interval Prevalence	Het aandeel van het meest voorkomende interval in een stuk.
Most Common Pitch Prevalence	Het aandeel van de meest voorkomende noot in een stuk.
Number of Common Melodic Intervals	Het aantal intervallen met een aandeel van minstens 9%.
Number of Common Pitches	Het aantal noten met een aandeel van minstens 9%.
Relative Strength of Most Common Intervals	Het aandeel van het 2de meest voorkomende interval gedeeld door het aandeel van het meest voorkomende interval.
Relative Strength of Top Pitch	Het aandeel van de 2de meest voorkomende noot gedeeld door het aandeel van de meest voorkomende noot.
Repeated Notes	Het relatieve aantal herhaalde noten in een stuk, oftewel MIDI nootintervallen van 0.
Size of Melodic Arcs	Het gemiddelde interval tussen melodische pieken en dalen.
Stepwise Motion	Het relatieve aantal intervallen met grootte 1 of 2.

Tabel 2: de gebruikte melodische jSymbolic statistieken en hun beschrijving.

7 Experimenten

7.1 Benaderingsstrategieën

In deel 2 werd besproken wat het soort muziek is dat we gaan genereren. In deel 4 werd besproken hoe we optimalisatieproblemen kunnen oplossen. Wat nog niet besproken is, is hoe we muziek als een probleem kunnen zien; als er iets geoptimaliseerd moet worden, moet er eerst een probleem worden vastgesteld. In dit deel wordt de gebruikte benaderingsstrategie besproken om het 'muziekprobleem' vast te leggen.

Er zijn quasi oneindig veel manieren om muziek te beschouwen. Muziek in zijn geheel werpt daardoor een onbehandelbaar probleem op. Bovendien blijken er geen eenvoudige regels te bestaan om neoklassieke gitaarmuziek te componeren. Om de grote complexiteit te reduceren blijven we daarom impliciete beperkingen invoeren tot het probleem aflijnbaar en dus behandelbaar wordt. Een eerste beperking in het muziekprobleem is om enkel de melodie te bekijken. Ritme, harmonie, tempo en verdere informatie wordt achterwege gelaten. Dit maakt het probleem al een stuk minder complex. Vervolgens wordt er naar een zienswijze gezocht die het muziekprobleem kadert en dus bijkomende restricties opwerpt. De eerste benaderingswijze is *de leerling-tovenaar benadering*. Muziek wordt gezien vanuit het perspectief van een onervaren aspirant-gitarist.

De leerling-tovenaar heeft weinig kennis of kunde en begint te oefenen op een klein en strikt bepaald aspect van neoklassieke gitaarmuziek. De enige basis vanwaar hij vertrekt zijn enkele muzikale invloeden die hij heeft. Ten eerste beperkt de leerling-tovenaar zich tot de natuurlijke mineurtoonladder in E_4, zogezegd de enige toonladder die hij beheerst. E_4 komt overeen met de hoogste open snaar van een in standaard E gestemde zessnarige gitaar. Dit betekent dat de leerling tovenaar maar acht bouwstenen gebruikt om een muziekstuk samen te stellen, namelijk de zeven noten van de natuurlijke mineurschaal in E_4 en de octaafnoot E_5. Bovendien zal hij zich focussen op een stijlkenmerk van neoclassical, namelijk orgelpunten. Om dit te doen baseert hij zich op een aantal invloeden. Concreet betekent dit dat er een aantal neoklassieke orgelpuntstukken gezocht worden die als referentie dienen voor de aspirant-gitarist. Van deze orgelpuntstukken worden de jSymbolic statistieken berekend die besproken werden in deel 6. De gemiddelden van deze statistieken worden afgewogen tegen de jSymbolic statistieken van de muziek die de leerling-tovenaar produceert. De variabelen die hieruit volgen vormen de basis van de doelfunctie en zijn dus de beslissingsvariabelen. De muziek die de leerling-tovenaar produceert moet zo goed als mogelijk aansluiten bij zijn invloeden. Met andere woorden, de afwijkingen tussen de gegenereerde statistieken en de referentiestatistieken moeten geminimaliseerd worden. Muziek wordt bijgevolg een minimalisatieprobleem. Dit optimalisatieprobleem wordt vervolgens opgelost met een Local Search algoritme. De leerling-tovenaarstrategie wordt in detail besproken in deel 7.2.

De aanpak van de leerling-tovenaar kent een aantal moeilijkheden. Ten eerste is het moeilijk om geschikte invloeden te vinden. Orgelpunten komen in overvloed voor in neoklassieke gitaarmuziek, maar zelden als het *enige* stijlkenmerk van een stuk muziek. Bovendien leek het nodig dat referentiestukken in mineur worden gespeeld (zie deel 7.2.1). De beperkingen die het muziekprobleem aflijnen werpen dus hun eigen problemen op in het zoeken naar referentiestukken. Uiteindelijk zijn er na uitgebreid zoeken maar een zevental stukken muziek gevonden die als referentie kunnen dienen. Ten tweede behelst deze aanpak maar één stijlkenmerk, namelijk orgelpunten. Voor andere stijlkenmerken zouden dan weer andere algoritmes moeten geschreven

worden. Deze algoritmes zouden vervolgens geïntegreerd moeten worden in een overkoepelend algoritme dat deze subalgoritmes in zijn geheel optimaliseert. Dit zou bijzonder archaïsch worden. Kortom, de leerling-tovenaarstrategie is globaal gezien niet bijzonder efficiënt. Dit gaf aanleiding tot een tweede aanpak, *de shredderbenadering*.

De shredderbenadering baseert zich op een enkele solo als invloed, de referentiesolo uit figuur 5. De metaforische *shredder* speelt improvisaties op de referentiesolo. Hij gebruikt dan de noten uit deze solo als bouwstenen om zelf een solo te maken en zou idealiter ook de stijlkenmerken kunnen spelen die in deze solo aanwezig zijn. De *shredder*algoritmes zouden dan zinvolle variaties moeten creëren op de referentiesolo. Deze benadering levert veel minder problemen op, aangezien er maar een input vereist is die bovendien minder restrictief is. Deze benadering wordt in detail besproken in deel 7.3.

7.2 De leerling-tovenaarstrategie

7.2.1 Het project *pedalpoints*

De Java code van het project *pedalpoints* is integraal terug te vinden in bijlage IV.

Het doel van dit project is om neoklassieke orgelpuntmuziek te genereren aan de hand van Local Search algoritmes en gebaseerd op de melodische kenmerken van enkele referentiestukken. Zoals besproken in de inleiding van dit deel wierp het zoeken naar deze referentiestukken enkele hindernissen op. Een eerste voorwaarde waaraan de referentiestukken moeten voldoen is dat ze bestaan uit orgelpuntmuziek. Zoniet, dan kunnen deze stukken niet dienen om orgelpuntmuziek te genereren. De lezer moet in het achterhoofd houden dat er geen expliciete compositieregels zijn gebruikt in dit werk. De referentiestukken zijn dus bepalend voor de aard van de gegenereerde melodieën. Referentiestukken die andere stijlkenmerken bevatten zoals bijvoorbeeld arpeggio's, zouden afbreuk kunnen doen aan het gezochte aspect orgelpunten. Dit wordt bevestigd door experimenten met het project *soloGen*. In dit project wordt er gebruik gemaakt van maar 1 referentiesolo dat alle stijlkenmerken omvat. Hierdoor verwatert een specifiek gezocht stijlkenmerk zoals orgelpunten in de gegenereerde oplossingen. Een tweede *veronderstelde* voorwaarde is dat deze referentiestukken gemaakt moeten zijn in een natuurlijke mineurschaal, aangezien de algoritmes enkel muziek genereren in deze schaal. De schaal van een stuk zou de waarden van de melodische jSymbolic statistieken kunnen beïnvloeden. Om dit aan te tonen of te ontkrachten is er een *Wilcoxon signed-ranks test* gedaan.

Wilcoxon signed-ranks test op de melodische aspecten van muziek

Er werden 22 stukken muziek verzameld om te gebruiken in de algoritmes van het project *pedalpoints*. De stukken zijn verdeeld in drie groepen, waarvoor van alle jSymbolic statistieken van deze stukken het gemiddelde is berekend per groep. De eerste groep bevat de gemiddeldes van de totale verzameling, genaamd *avgglobal*. De tweede groep bevat de gemiddeldes van acht stukken waarvan de auteur zeker weet dat ze in een natuurlijke mineurschaal zijn geschreven, genaamd *avgminor*. de derde groep bestaat uit gemiddeldes van 6 stukken waarvan de auteur weet dat ze zeker niet in een natuurlijke mineurschaal zijn geschreven, genaamd *avgnotminor*. Een stukje van deze dataset is te zien in tabel 3.

jSymbolic statistieken	***avgglobal***	***avgminor***	***avgnotminor***
Amount of arpeggiation	0,476745	0,542288	0,381617
Average melodic interval	5,468682	5,831125	4,944
Chromatic motion	0,097632	0,05527	0,15391
Direction of motion	0,5042	0,521725	0,469017

Tabel 3: staal van de dataset.

Het valt op dat de gemiddelde jSymbolic waarden per groep nogal sterk verschillen. Om na te gaan of deze groepen werkelijk anders zijn, wordt er een Wilcoxon signed-ranks test gedaan. Deze toets wordt gebruikt voor datasets waarvoor geen normaliteit verondersteld kan worden en test onder de nulhypothese of de mediaan van twee datasets hetzelfde is (Moore, McCabe en Craig, 2007). Er wordt verwacht dat de groepen sterk verschillen van elkaar en dat de nulhypothese verworpen wordt. De test werd uitgevoerd in SPSS, waarvan de output te vinden is in figuur 11.

Ranks

		N	Mean Rank	Sum of Ranks
avgminor - avgglobal	Negative Ranks	17[a]	15,15	257,50
	Positive Ranks	12[b]	14,79	177,50
	Ties	1[c]		
	Total	30		
avgnotminor - avgglobal	Negative Ranks	11[d]	11,05	121,50
	Positive Ranks	18[e]	17,42	313,50
	Ties	1[f]		
	Total	30		
avgnotminor - avgminor	Negative Ranks	13[g]	11,92	155,00
	Positive Ranks	15[h]	16,73	251,00
	Ties	2[i]		
	Total	30		

a. avgminor < avgglobal
b. avgminor > avgglobal
c. avgminor = avgglobal
d. avgnotminor < avgglobal
e. avgnotminor > avgglobal
f. avgnotminor = avgglobal
g. avgnotminor < avgminor
h. avgnotminor > avgminor
i. avgnotminor = avgminor

Test Statistics[a]

	avgminor - avgglobal	avgnotminor - avgglobal	avgnotminor - avgminor
Z	-,865[b]	-2,076[c]	-1,093[c]
Asymp. Sig. (2-tailed)	,387	,038	,274

a. Wilcoxon Signed Ranks Test
b. Based on positive ranks.
c. Based on negative ranks.

Figuur 11: SPSS output 1.

De test begint met het paarsgewijze verschil te nemen tussen twee groepen van gegevens. Dan worden de absolute verschillen van de gegevensparen gerangschikt van klein naar groot. Indien twee absolute verschillen even groot zijn, dan krijgen ze het gemiddelde van de respectievelijk opeenvolgende rangen toegewezen. Vervolgens worden de rangen van de negatieve verschillen bij elkaar opgeteld en hetzelfde gebeurt voor de rangen van de positieve verschillen. Voor twee gelijkaardige groepen zou de som van de positieve rangen ongeveer gelijk moeten zijn aan de som van de negatieve rangen. Bijgevolg wordt de nulhypothese verworpen wanneer de som van de positieve rangen sterk verschilt van de som van de negatieve rangen. De verwerping gebeurt wanneer de kleinste som van de twee rangsoorten kleiner is dan een kritieke waarde. Een alternatieve verwerpingtoets kan gedaan worden aan de hand van de door SPSS berekende *p* waarde

(Hole, 2011). Voor een significantieniveau van 5% en met 29 getelde rangparen[1] komt de kritieke waarde op 126 te liggen (Zaiontz, 2014). Uit de SPSS output 1 blijkt dat enkel voor het paar *avgnotminor-avgglobal* de nulhypothese verworpen wordt. Voor de groepsparen *avgminor-avgglobal* en *avgnotminor-avgminor* wordt de nulhypothese niet verworpen. Van deze groepen kan dus niet gezegd worden dat ze paarsgewijs sterk verschillen.

Merk op dat de test gebruik maakt van 30 jSymbolic gegevens en dus meer dan alleen de 21 die geselecteerd zijn voor de algoritmes. De Wilcoxon test wordt opnieuw gedaan, maar dan enkel voor de jSymbolic gegevens die ten eerste bij de geselecteerde 21 zitten en die ten tweede beïnvloed zouden kunnen worden door de toonladder van een muziekstuk. Bepaalde van deze jSymbolic statistieken kunnen logischerwijze zeker niet door de toonladder beïnvloed worden. Een voorbeeld hiervan is *Repeated Notes*. Deze statistiek geeft het relatieve aantal herhaalde noten weer, oftewel de fractie van intervallen met waarde 0. Het is evident dat het aantal herhaalde noten in een stuk volledig losstaat van de gebruikte toonladder.

Ranks

		N	Mean Rank	Sum of Ranks
avgminor - avgglobal	Negative Ranks	7[a]	5,00	35,00
	Positive Ranks	4[b]	7,75	31,00
	Ties	0[c]		
	Total	11		
avgnotminor - avgglobal	Negative Ranks	4[d]	5,75	23,00
	Positive Ranks	7[e]	6,14	43,00
	Ties	0[f]		
	Total	11		
avgnotminor - avgminor	Negative Ranks	5[g]	5,80	29,00
	Positive Ranks	6[h]	6,17	37,00
	Ties	0[i]		
	Total	11		

a. avgminor < avgglobal
b. avgminor > avgglobal
c. avgminor = avgglobal
d. avgnotminor < avgglobal
e. avgnotminor > avgglobal
f. avgnotminor = avgglobal
g. avgnotminor < avgminor
h. avgnotminor > avgminor
i. avgnotminor = avgminor

[1] Paarsgewijze verschillen van 0 worden uit de test gehaald. Op 30 gegevensparen komt dit 1 keer voor bij alle groepsparen.

Test Statistics[a]

	avgminor - avgglobal	avgnotminor - avgglobal	avgnotminor - avgminor
Z	-,178[b]	-,889[c]	-,356[c]
Asymp. Sig. (2-tailed)	,859	,374	,722

a. Wilcoxon Signed Ranks Test

b. Based on positive ranks.

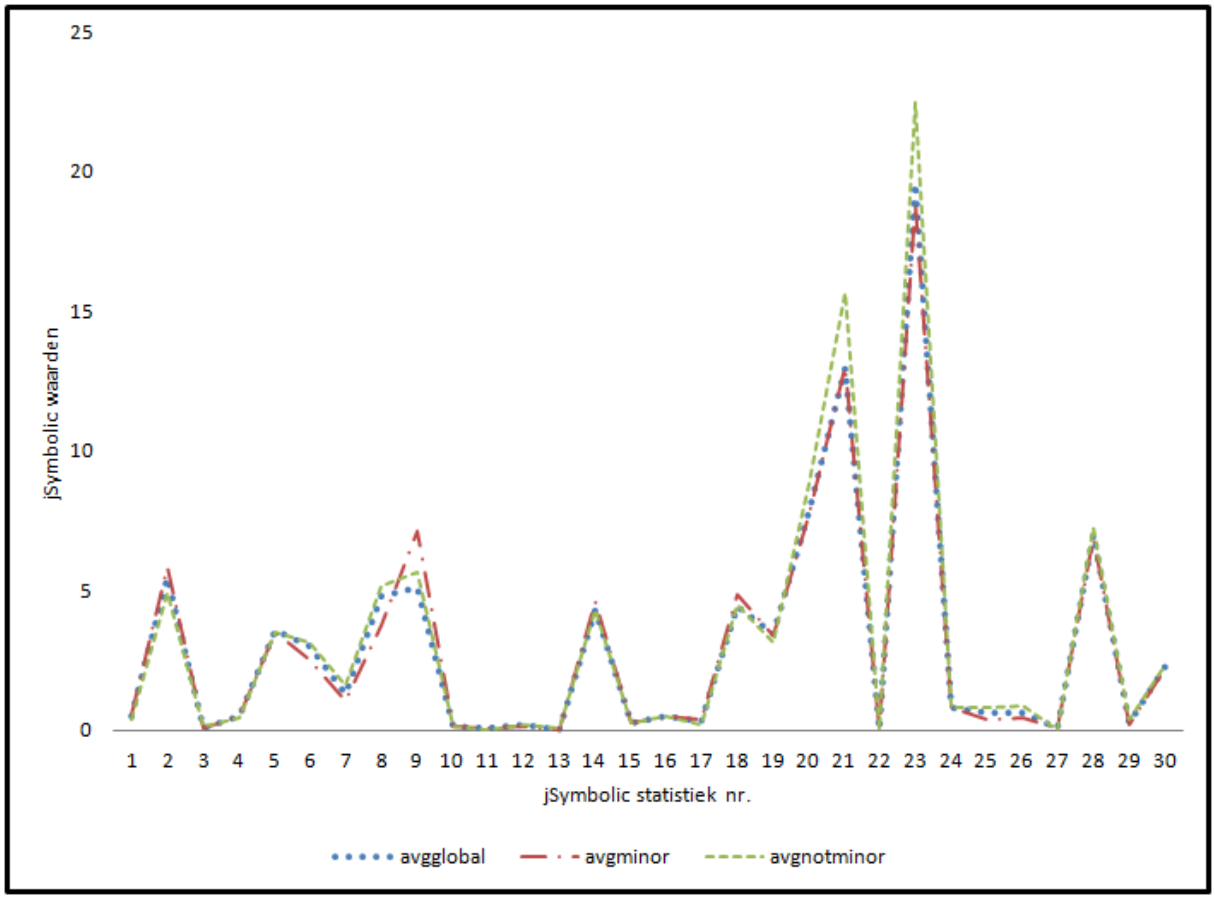
c. Based on negative ranks.

Figuur 12: SPSS output 2.

Zoals te zien is in de SPSS output van figuur 12, wordt de nulhypothese in geen van de gevallen verworpen. Voor de 11 gekozen jSymbolic statistieken zijn er geen significante verschillen te merken tussen de groepen.

Conclusie van de test

De verwachting dat toonladders een invloed hebben op de jSymbolic statistieken was voorbarig. Er kan niet aangetoond worden dat het gebruik van verschillende toonladders leidt tot significant andere waarden van deze jSymbolic statistieken, aangezien de nulhypothese zelden en bovendien inconsistent verworpen wordt. Een visuele voorstelling van de gegevens toont dat de groepen onderling minder verschillen dan op het eerste zicht leek.



Figuur 13: grafische weergave van de gebruikte gegevens in de Wilcoxon testen.

Desalniettemin wordt het resultaat van deze testen voor de zekerheid genegeerd en zal er enkel gebruik gemaakt worden van referentiestukken in een natuurlijke mineurschaal, teneinde de geschiktheid van de referentiegegevens optimaal te houden.

7.2.1.1 De eerste doelfunctie

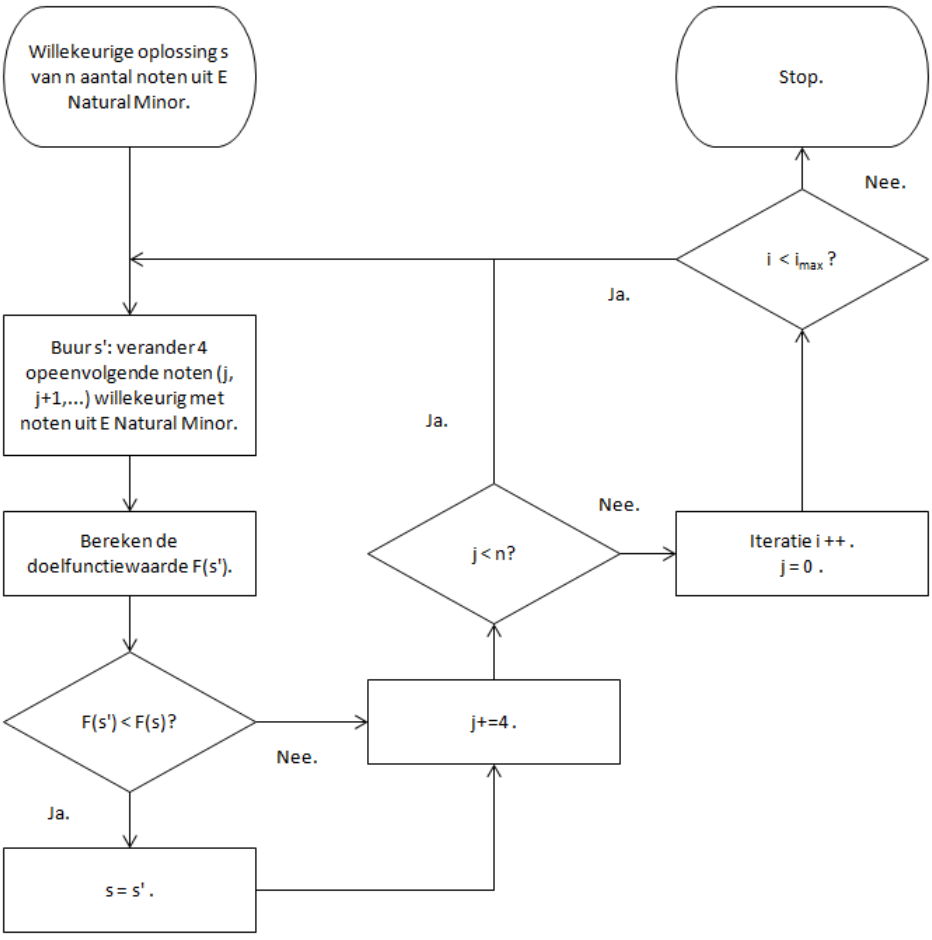
De doelfunctie in het project *pedalpoints* bestaat uit de sommatie van een aantal jSymbolic gebaseerde variabelen. Aangezien een gegenereerd stuk zo nauw mogelijk moet aansluiten aan de referentiewaarden wordt ervoor geopteerd om de procentuele afwijking te minimaliseren.

$$F(s) = \min\left\{\sum_{i=1}^{21} \frac{x_i - y_i}{y_i}\right\} \tag{7.1}$$

In formule 7.1 is x_i de waarde van jSymbolic statistiek *i* voor een algoritmisch gegenereerd stuk. Er zijn er zo 21. Van deze waarde wordt de procentuele afwijking berekend tegenover referentiewaarde y_i. Deze waarde y_i is het gemiddelde van de waarden van de referentiestukken voor statistiek *i*. Deze doelfunctie is terug te vinden in bijlage IV.11.

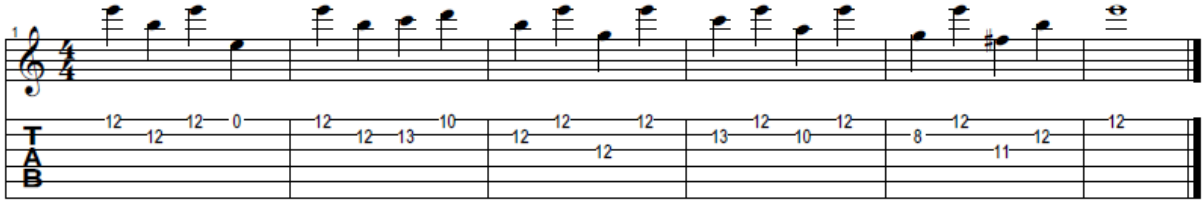
7.2.1.2 Algoritme 1

Figuur 14 geeft een process flow chart weer van het eerste algoritme van het project. De Java code voor dit algoritme is terug te vinden in bijlage IV.2.

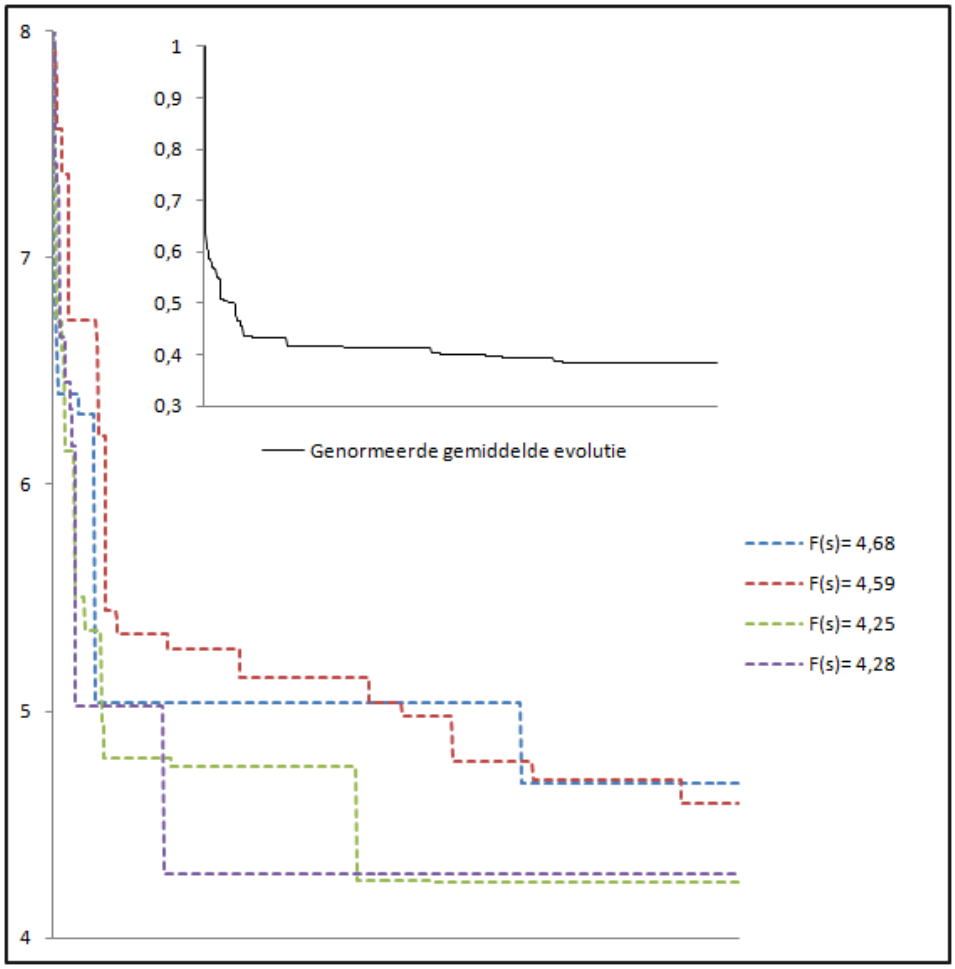


Figuur 14: een process flow chart van *Algo1*.

Algo1 is een eenvoudig Local Search algoritme. Het algoritme begint met het maken van een willekeurige oplossing van *n* noten uit de natuurlijke mineurtoonladder van E_4. Vervolgens wordt er een buur *s'* gegenereerd. De buur verschilt in 4 opeenvolgende noten van de huidige oplossing, willekeurig geselecteerd uit de natuurlijke mineurtoonladder van E_4. Van deze buur wordt de waarde van de doelfunctie (7.1) berekend. Is deze waarde kleiner dan de doelfunctiewaarde van de huidige oplossing, dan wordt de buur aanvaard als de nieuwe huidige oplossing. *Algo1* is dus een first improving LS algoritme. Daarna wordt er overgegaan naar de vier volgende noten in de oplossing en begint de procedure van voor af aan. Wanneer alle *n* noten een keer doorlopen zijn, eindigt de iteratie en begint de volgende iteratie. Na een groot aantal iteraties resulteert het algoritme in een melodisch stukje muziek. Een voorbeeld van een orgelpuntstuk van 20 noten gegenereerd door *Algo1* is te zien in figuur 15. Voor de duidelijkheid is het orgelpunt in de laatste maat nog eens toegevoegd als een hele noot.



Figuur 15: een stuk neoklassieke orgelpuntmuziek gegenereerd door *Algo1*.



Figuur 16: de evolutie van enkele doelfunctiewaarden van *Algo1* voor 20 noten en 1000 iteraties.

Bedenkingen bij algoritme 1

Algo1 werkt behoorlijk goed en is bovendien snel. De gegenereerde stukken zijn effectief orgelpuntstukken en klinken melodisch. Subjectief zou het weliswaar omschreven kunnen worden als een beetje 'slordig'. Als we de evolutie van de doelfunctiewaarde bekijken, dan vlakt deze consistent redelijk snel af. Figuur 16 geeft deze evolutie weer voor 4 gegenereerde voorbeelden. Van deze vier is tevens het genormeerde gemiddelde berekend. Dit leert ons dat het algoritme de doelfunctiewaarde reduceert tot ongeveer 40% ten opzichte van de doelfunctiewaarde van de eerste gegenereerde oplossing.

Aangezien *Algo1* conceptueel erg eenvoudig is, zijn er nog enkele aanpassingen aan te bedenken. Dit heeft geleid tot de ontwikkeling van *Algo1v2*.

7.2.1.3 Algoritme 1 versie 2

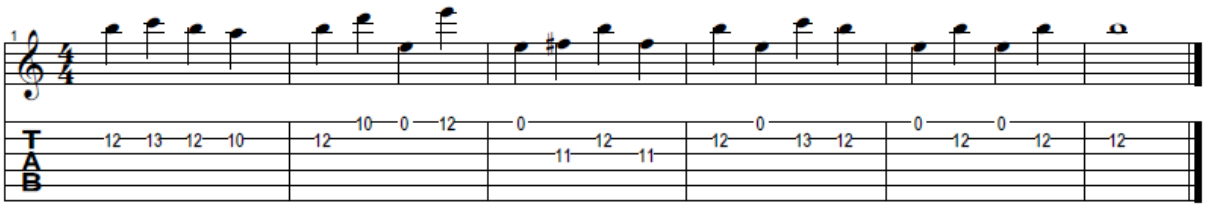
Algo1v2 bouwt verder op *Algo1*. De Java code voor dit algoritme is integraal terug te vinden in bijlage IV.6.

In tegenstelling tot *Algo1* maakt *Algo1v2* wel gebruik van verschillende buren. Er worden drie buren gegenereerd, namelijk s_2', s_3' en s_4'. Deze buren vervangen respectievelijk 2, 3 en 4 opeenvolgende elementen in de huidige oplossing, willekeurig gekozen uit de natuurlijke mineurschaal van E_4. Van deze buren worden de doelfunctiewaarden $F(s_2')$, $F(s_3')$ en $F(s_4')$ berekend. De kleinste van deze doelfunctiewaarden $F(s_k')$ wordt afgewogen tegen de doelfunctiewaarde van de huidige oplossing $F(s)$. Indien $F(s_k')$ kleiner is dan $F(s)$, dan wordt oplossing s_k' aanvaard als de nieuwe beste oplossing. *Algo1v2* is dus een soort van steepest descent LS algoritme.

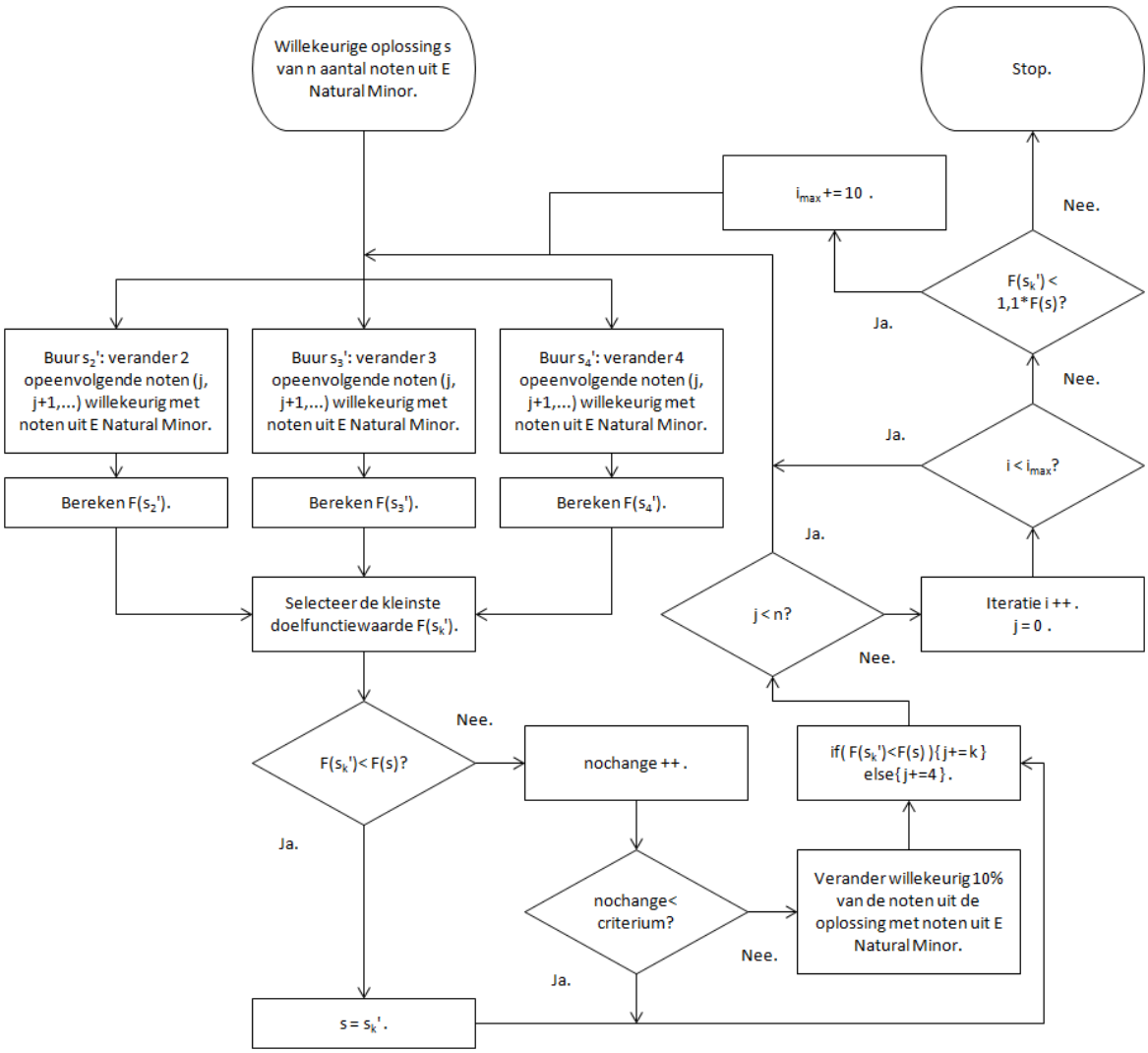
Bijkomend is er een perturbatieregel toegevoegd aan het algoritme. Indien er na een aantal iteraties geen nieuwe oplossing gevonden wordt, dan worden er willekeurig 10% van de noten in de huidige oplossing vervangen. De huidige oplossing wordt voor de perturbatie opgeslagen in een geheugen. Het criterium dat overschreden dient te worden staat in functie van het aantal noten. Proefondervindelijk is vastgesteld dat het optimaal is om dit criterium kleiner te maken voor oplossingen met veel noten.

Tot slot staat het aantal iteraties in functie van de kleinste doelfunctiewaarde. Indien de laatst gevonden doelfunctiewaarde kleiner is dan de kleinste doelfunctiewaarde vermenigvuldigd met een multiplicator (bv. 1,1), dan wordt het aantal iteraties automatisch vermeerderd. Op den duur zal het algoritme hoe dan ook stoppen, aangezien de kleinste tot dan toe gevonden doelfunctiewaarde steeds kleiner wordt. Ook zullen er na verloop van tijd steeds meer perturbaties voorkomen. Hierdoor zal het criterium steeds minder kans hebben om overschreden te worden. Op deze manier kan het aantal iteraties dynamisch ingebouwd worden, teneinde een optimaal aantal iteraties te garanderen.

De process flow chart in figuur 18 geeft de werking van algoritme 1 versie 2 conceptueel weer.



Figuur 17: een orgelpuntstuk gegenereerd met *Algo1v2*.



Figuur 18: een process flow chart van *Algo1v2*.

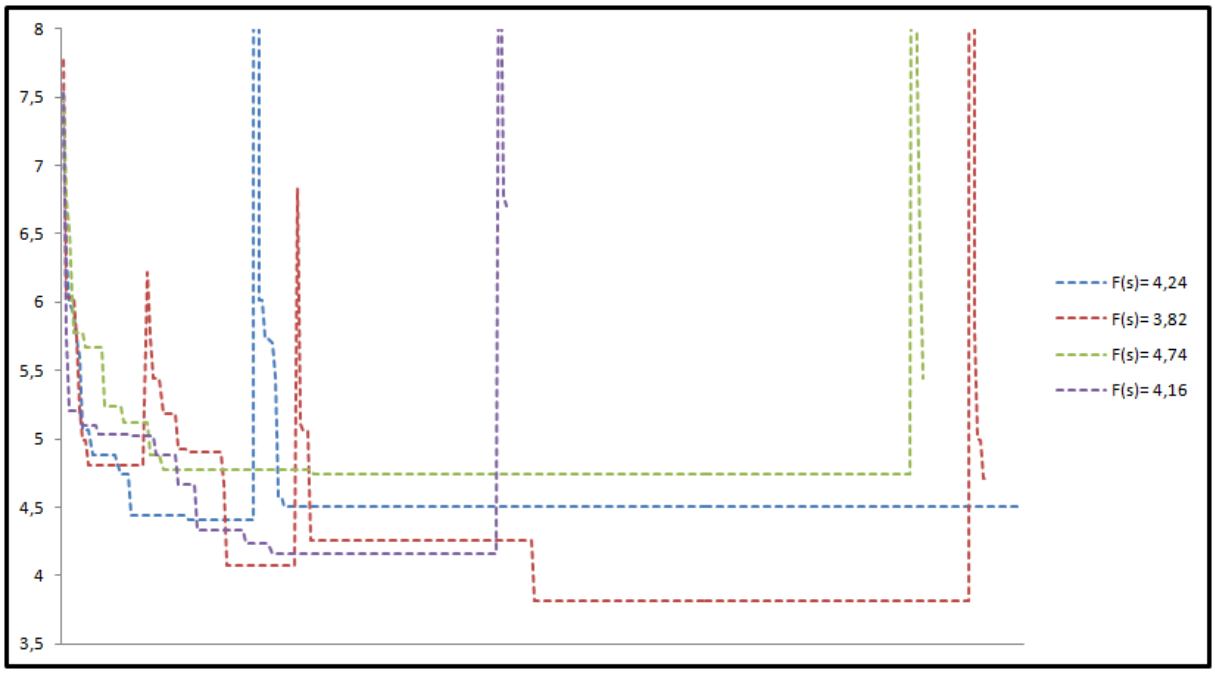
Bedenkingen bij Algoritme 1 versie 2

Algo1v2 levert iets betere resultaten dan *Algo1* afgaande op de doelfunctiewaarden. Deze liggen een beetje lager en worden doorgaans in heel wat minder iteraties bereikt. Subjectief gezien gaat de muzikale kwaliteit er echter weinig op vooruit. De gegenereerde muziekstukken zijn melodische orgelpuntmuziek, maar klinken nog steeds ietwat chaotisch. Dit gaf aanleiding tot het ontwikkelen van algoritme 2.

Figuur 19 geeft de evolutie van de doelfunctiewaarden weer voor een aantal voorbeelden. Merk op dat er hevige pieken optreden in deze evoluties. Deze pieken gebeuren na perturbaties. Het optimaal voorkomen van perturbaties moet experimenteel bepaald worden, hetgeen niet eenduidig te bepalen is. Op de figuur is bijvoorbeeld te zien dat de rode lijn op zijn laagste punt komt na 2 perturbaties. De paarse lijn komt echter op zijn laagste punt voor de eerste perturbatie.

Kwalitatief gezien produceert *Algo1v2* ongeveer gelijkwaardige oplossingen als *Algo1*. *Algo1v2* doet dit echter in heel wat minder iteraties. Zo heeft het paarse voorbeeld maar 130 iteraties nodig om een goede oplossing te genereren, terwijl de voorbeelden van *Algo1* steevast op 1000 iteraties werken. Dit komt door de dynamische instelling van het algoritme: indien er een goede oplossing

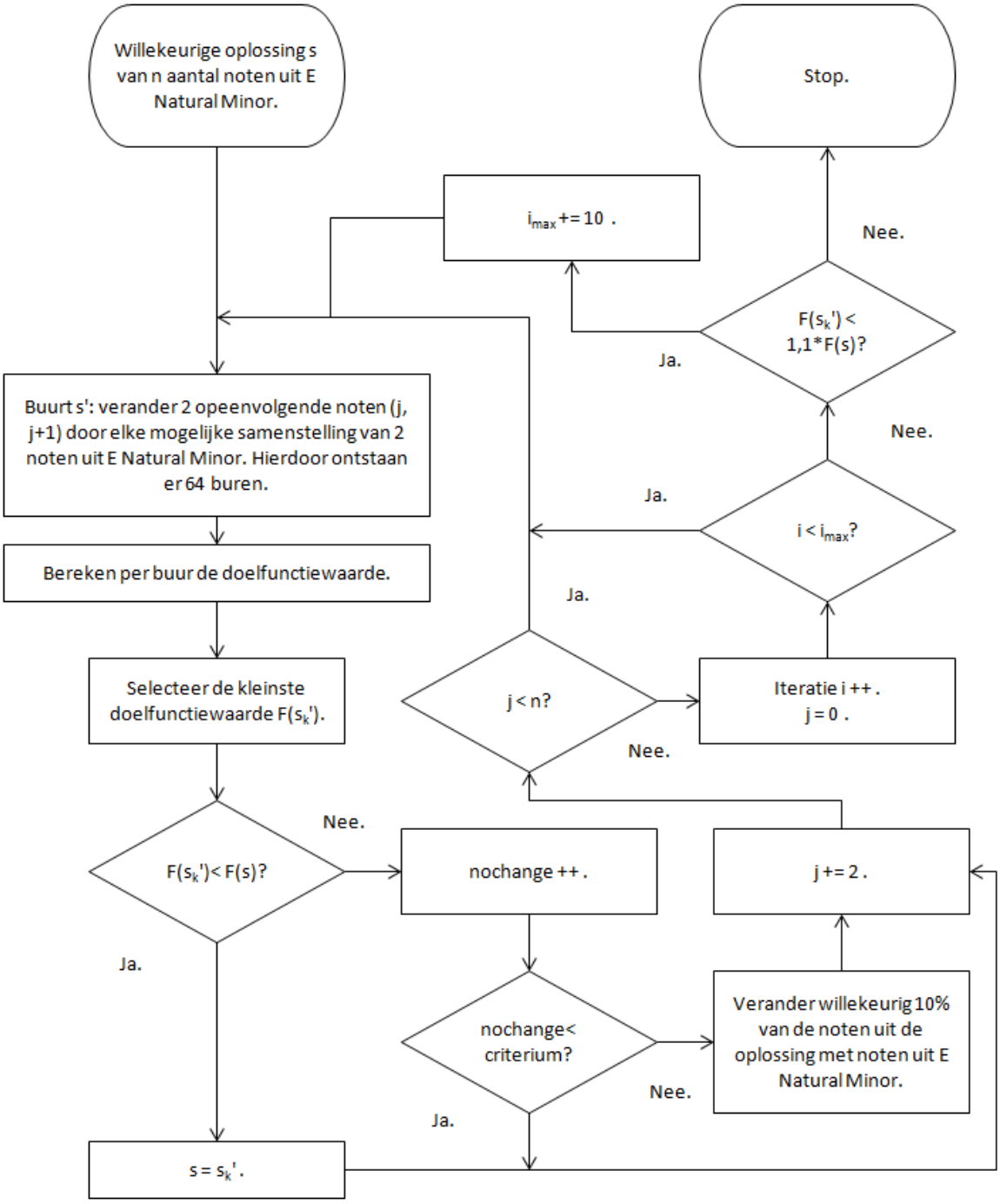
gevonden is, wordt de uitvoering van het algoritme niet nodeloos lang gerekt. Een goede oplossing wordt ook sneller bekomen door verschillende buren onderling af te wegen.



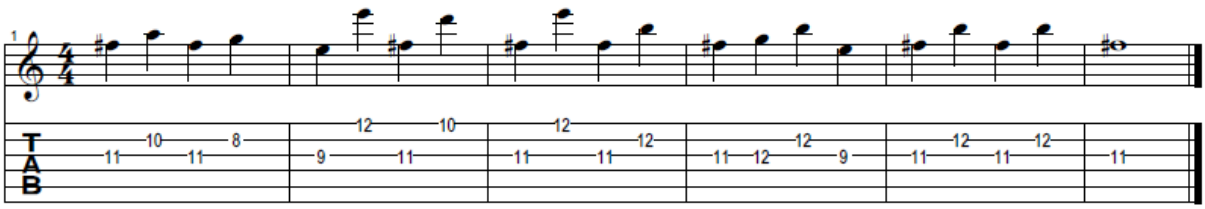
Figuur 19: de evolutie van enkele doelfunctiewaarden van *Algo1v2* voor 20 noten en 130 (paars) tot 810 (blauw) iteraties. Minimum aantal iteraties vastgelegd op 100.

7.2.1.4 Algoritme 2 en 2v2

Een volgende stap in het zoeken naar een goed Local Search algoritme is het aanpakken van de willekeurige generatie van buren. Algoritme 2 is identiek aan algoritme 1 met het verschil dat er in plaats van 4 opeenvolgende noten willekeurig worden veranderd, alle mogelijke samenstellingen van 4 noten onderling worden afgewogen. Praktisch gezien bleek *Algo2* echter onuitvoerbaar. De traagste stap in de algoritmes van project *pedalpoints* is het berekenen van de doelfunctie. De jSymbolic statistieken waaruit deze doelfunctie bestaat worden algoritmisch berekenend en geïtereerd aan de hand van het aantal noten in een oplossing. Voor oplossingen met een groot aantal noten komt dit neer op erg veel stappen om een enkele doelfunctiewaarde te berekenen. Als men nu elke mogelijke samenstelling van vier noten uit de natuurlijke mineurschaal onderling wil afwegen, dan moet men 8^4= 4096 doelfunctiewaarden berekenen. Dit neemt veel tijd in beslag, waardoor *Algo2* extreem traag is. *Algo2v2* pakt dit aan door alle mogelijke samenstellingen van twee opeenvolgende noten af te wegen. Hierdoor moeten er maar 8^2= 64 buren onderling worden afgewogen in plaats van 4096. Een process flow chart van *Algo2v2* is te zien in figuur 20. De Java code van het algoritme is terug te vinden in bijlage IV.8.



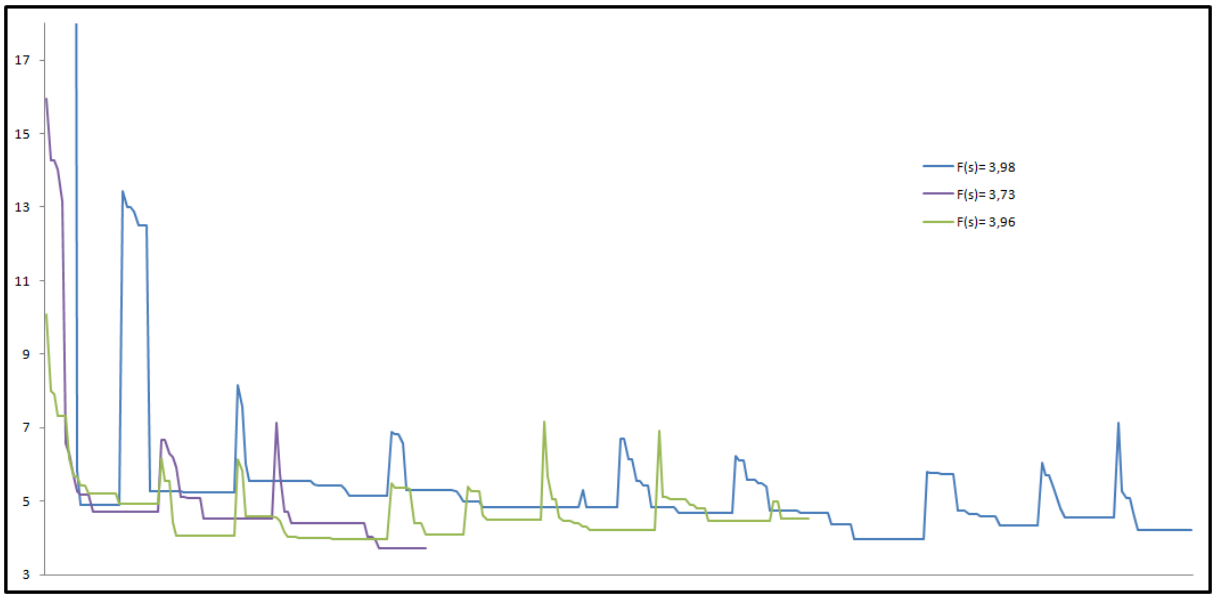
Figuur 20: process flow chart van *Algo2v2*.



Figuur 21: een orgelpuntstuk gegenereerd met *Algo2v2*.

Bedenkingen bij Algo2v2

Tegenover het eerste algoritme, bekomt *Algo2v2* een lagere doelfunctiewaarde in veel minder iteraties. Dit voordeel wordt echter gecompenseerd door de langere executietijd van het algoritme. Voor oplossingen met weinig noten is het verschil in executietijd weinig merkbaar (<40 noten), maar voor oplossingen met veel noten wel (>80 noten). Figuur 22 geeft de evolutie van de doelfunctiewaarde weer voor enkele voorbeelden van *Algo2v2*.



Figuur 22: de evolutie van enkele doelfunctiewaarden van *Algo2v2* voor 20 noten en 10 (paars) tot 30 (blauw) iteraties. Minimum aantal iteraties vastgelegd op 10.

Alhoewel de objectieve performantie van *Algo2v2* beter is dan voorgaande algoritmes, voegt het subjectief gezien niets tot weinig toe aan de muzikale kwaliteit van de stukken dat het genereert. Er moet dus een andere manier gezocht worden om goede muziek te genereren. Hiertoe is er een Markov gebaseerde foutscore toegevoegd aan de doelfunctie, teneinde 'chaotische' sprongen in de oplossing te verminderen.

7.2.1.5 Aanpassing van de doelfunctie

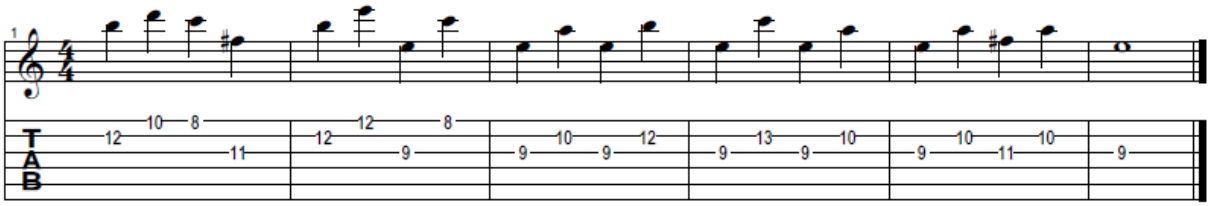
De doelfunctie wordt aangepast om het resultaat van de algoritmes te manipuleren. De bedoeling is om subjectief betere muziekstukken te genereren. Gegenereerde stukken uit voorgaande delen hadden iets 'willekeurigs' of 'chaotisch'. Om dit te verminderen is er een term toegevoegd aan de doelfunctie. Deze term voegt een foutscore toe aan de doelfunctiewaarde, indien er noottransities worden gekozen met een lage Markov transitiekans. Deze transitiekansen zijn afgeleid van de melodische intervallen uit de referentiestukken. Hoe minder waarschijnlijk de overgang van de ene noot naar de andere is, hoe hoger deze strafscore wordt. Aan de betreffende algoritmes wordt de notatie *CEM* of *CEM2* toegevoegd. *CE* staat voor *Cross Entropy*, het principe waarop de strafscore is gebaseerd (Wikipedia, 2014). *M* en *M2* staat respectievelijk voor 1ste orde Markov transitiekansen en 2de orde Markov transitiekansen. Een 2de orde Markov transitiekans is de kans op een gebeurtenis gegeven het voorkomen van 2 voorgaande gebeurtenissen. Dit in tegenstelling tot een eerste orde transitiekans, die enkel 1 voorgaande gebeurtenis beschouwt (Cogill, 2011). De formule voor *DoelfunctieCEM2* ziet er uit als volgt:

$$F(s) = \min \left\{ \begin{array}{c} \sum_{i=1}^{21} \frac{x_i - y_i}{y_i} \\ + \sum_{i=2}^{n} log_{10}\left(\frac{1}{P(S_i = s_i | \, S_{i-1} = s_{i-1})}\right) \\ + \sum_{i=3}^{n} log_{10}\left(\frac{1}{P(S_i = s_i | \, S_{i-1} = s_{i-1}, S_{i-2} = s_{i-2})}\right) \end{array} \right\} \quad (7.2)$$

Merk op dat $P(S_i = s_i | \, S_{i-1} = s_{i-1})$ een andere manier is om de transitiekans p_{ij} tussen de toestanden of gebeurtenissen s_i en s_j te schrijven, zoals gedaan werd in deel 5. De letter n boven de sommatietekens staat voor het aantal noten in de oplossing. De functie $y = log(1/x)$ wordt gebruikt omdat deze de juiste eigenschappen heeft. Voor $x \to 0$ gaat $y \to \infty$ en voor $x \to 1$ gaat $y \to 0$. In mensentaal: kansen die groot zijn zullen resulteren in kleine strafscores. De Java code voor deze doelfunctie is terug te vinden in bijlage IV.13.

7.2.1.6 Algo2v2CEM2

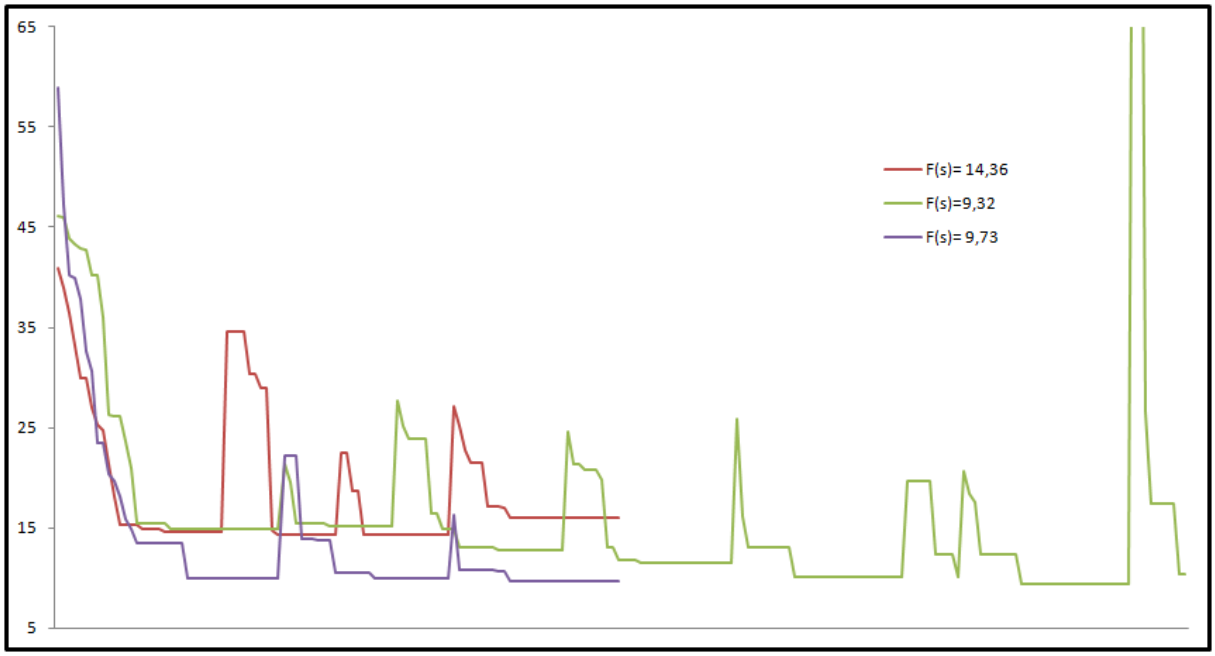
Met de nieuwe doelfunctie zijn er verschillende experimenten gedaan. Aangezien *Algo2v2CEM2* daarvan de beste is, zal enkel *Algo2v2CEM2* besproken worden. Dit algoritme is quasi identiek aan *Algo2v2*, met het grote verschil dat het gebruik maakt van doelfunctie (7.2).



Figuur 23: een stuk gegenereerd door *Algo2v2CEM2*.

Bedenkingen bij Algo2v2CEM2

In se is *Algo2v2CEM2* hetzelfde algoritme als *Algo2v2*. Het verschil huist in het gebruik van een andere doelfunctie, namelijk een ook die gebruik maakt van Markov transitiekansen. Hierdoor klinken de gegenereerde muziekstukken subjectief gezien iets beter. De Markov transitiekansen en de jSymbolic statistieken worden geschat aan de hand van meerdere referentiestukken. De thema's van deze stukken verwateren hierdoor weliswaar, omdat de karakteristieken van de stukken tezamen worden genomen. De gegenereerde stukken klinken dan ook (nog steeds) volledig themaloos. In het volgende project genaam *soloGen* worden stukken gegenereerd aan de hand van 1 referentiesolo, in een poging de unieke eigenschappen van het referentiestuk beter terug te horen in de gegenereerde oplossingen. In *soloGen* (zie deel 7.3) wordt vertrokken van het beste algoritme uit het project *pedalpoints*, namelijk *Algo2v2CEM2*.



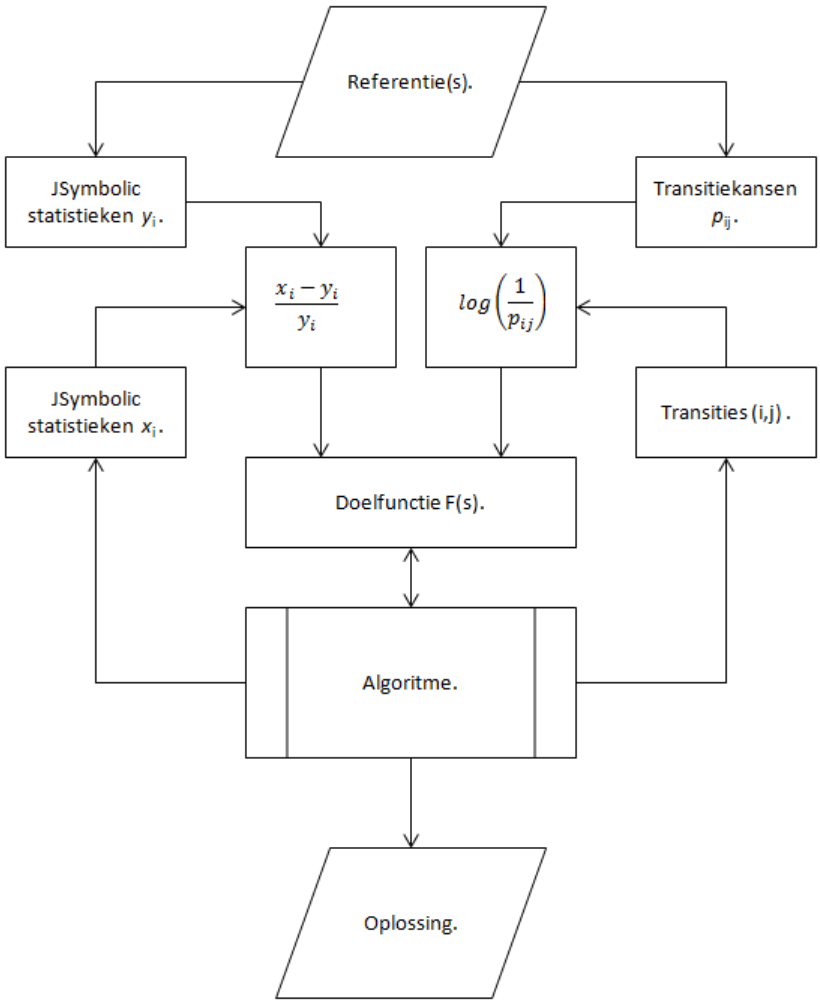
Figuur 24: de evolutie van enkele doelfunctiewaarden van *Algo2v2CEM2*.

7.3 De shredderbenadering

7.3.1 Het project soloGen

7.3.1.1 Overzicht van het project

Het project soloGen bouwt voort op het project *pedalpoints*. Op een aantal aanpassingen na zijn de twee projecten gelijkaardig qua opbouw. Ter overzicht geeft figuur 25 een robuuste process flow chart weer van de structuur van beide projecten. Het project *soloGen* is volledig terug te vinden in bijlage V.



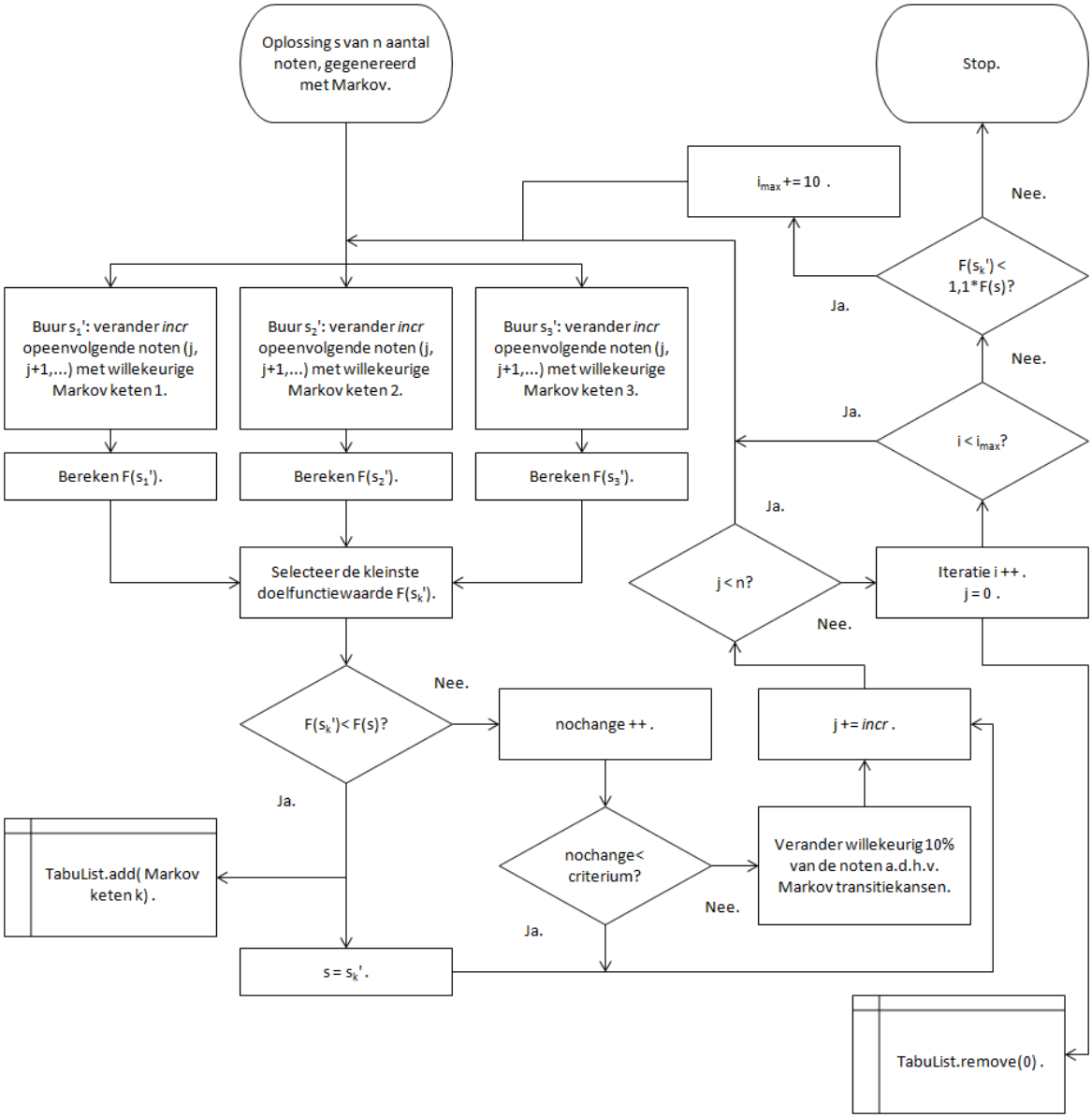
Figuur 25: process flow chart van de projecten *pedalpoints* en *soloGen*.

Het grote verschil tussen beide projecten zit in het gebruik van de referenties. Het project *pedalpoints* gebruikt meerdere referenties om orgelpuntmuziek te creëren. Het project *soloGen* gebruikt maar 1 referentie(solo) en zou solo's moeten produceren die nauw aansluiten bij deze referentie. De eerste experimenten uit *soloGen* gaan voort op de algoritmes uit *pedalpoints*, in de eerste plaats *Algo2v2CEM2*. Dit gaf echter wat problemen. Zoals reeds besproken heeft *Algo2v2CEM2* een redelijk lange executietijd nodig, die bovendien groeit voor grote oplossingen. In *soloGen* worden er oplossingen gegenereerd die even groot zijn als de referentiesolo, waarvoor *Algo2v2CEM2* te traag is. Merk op dat de referentiesolo uit figuur 5 bestaat uit 146 noten en 19

unieke elementen. Voor de referentiesolo zou het algoritme dan telkens 19^2= 361 buren moeten maken per 2 noten die veranderd worden. Dit is praktisch gezien niet realiseerbaar en heeft geleid tot de ontwikkeling van Algoritme 3.

7.3.1.2 Algoritme 3

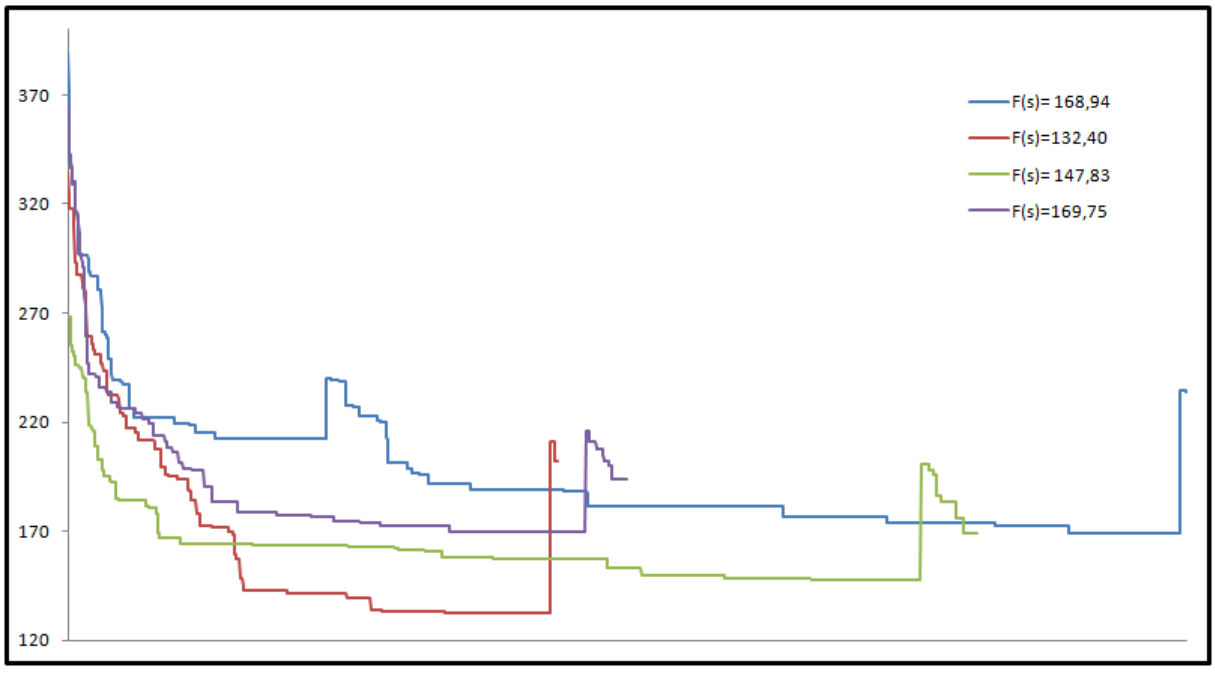
Algo3 is een Local Search algoritme en bouwt voort op de algoritmes uit het project *pedalpoints*. In plaats van alle mogelijke samenstellingen van noten te beschouwen of noten willekeurig te veranderen, worden groepen van noten uit de oplossing vervangen door kleine Markov ketens gebaseerd op de referentiesolo. Door grote delen uit de oplossing tegelijk te veranderen (bijvoorbeeld 5%) en door maar enkele buren onderling af te wegen, komt het algoritme in relatief weinig tijd tot een goede oplossing. Een process flow chart van het algoritme wordt weergegeven in figuur 26. De Java code voor *Algo3* is integraal terug te vinden in bijlage V.3.



Figuur 26: een process flow chart van *Algo3*.

Algo3 maakt gebruik van doelfunctie (7.2) en bouwt oplossingen met elementen uit de referentiesolo. Het algoritme wordt geïnitialiseerd met het aanmaken van een Markov keten van *n* noten, gebaseerd op transitiekansen geschat aan de hand van de referentiesolo. Vervolgens worden er drie buren gegenereerd door een groep van opeenvolgende noten te vervangen met Markov ketens. Deze ketens worden geïnitialiseerd op het element vlak voor de groep met te vervangen noten. Hierdoor zouden de ingevoegde ketens mooi moeten aansluiten op de rest van de oplossing. De buren worden onderling en tegen de huidige oplossing afgewogen, waarna de beste wordt overgehouden. Indien een buur als nieuwe oplossing wordt gekozen, dan wordt het stukje Markov keten dat deze buur onderscheidt van de oorspronkelijke oplossing in een tabu list gestoken. Dit wordt gedaan om de variatie in de uiteindelijke oplossing zo groot mogelijk te houden. Daarna wordt overgegaan op de volgende groep noten. Wanneer alle noten uit de globale oplossing doorlopen zijn, wordt er overgegaan naar de volgende iteratie en wordt de oudste keten terug uit de tabu list gehaald. Dit gebeurt om 2 redenen. Langs de ene kant om goede stukken keten toch meerdere keren in een oplossing toe te laten en langs de andere kant om de tabu list niet te groot te laten worden. Hoe groter deze lijst, hoe langer het programma erover doet om door de lijst te zoeken. Een lijst die heel groot wordt, vertraagt het programma aanzienlijk.

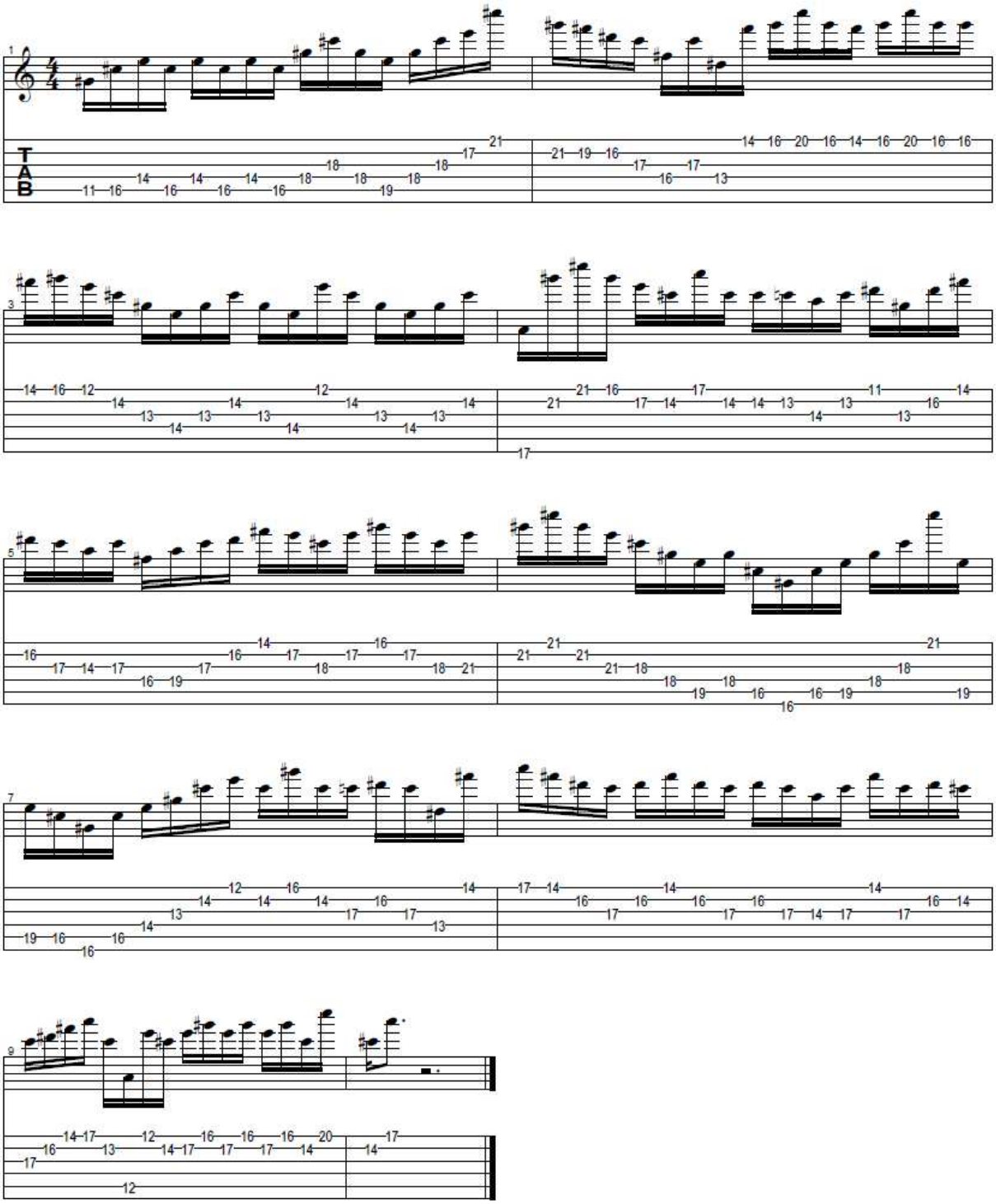
Merk tevens op dat ook aan dit algoritme een perturbatieregel en een dynamisch aantal iteraties is toegevoegd. De oude perturbatieregel, die willekeurig 10% van noten verandert door andere willekeurig gekozen noten, kan hier niet gebruikt worden. Deze regel resulteert in *Algo3* in rare sprongen die het algoritme er niet altijd terug uithaalt. Figuur 28 geeft een gegenereerde solo weer die over het algemeen goed klinkt afgezien van enkele vreemde sprongen, zoals bijvoorbeeld te zien is in het begin van maat vier. Door ook bij de perturbatie rekening te houden met Markov transitiekansen komen deze chaotische elementen niet meer voor in het resultaat. Er worden dan willekeurig 10 noten veranderd in functie van de noot die ervoor komt. Dit resulteert in oplossingen zonder vreemde melodische sprongen, waarvan een voorbeeld te zien is in figuur 29.



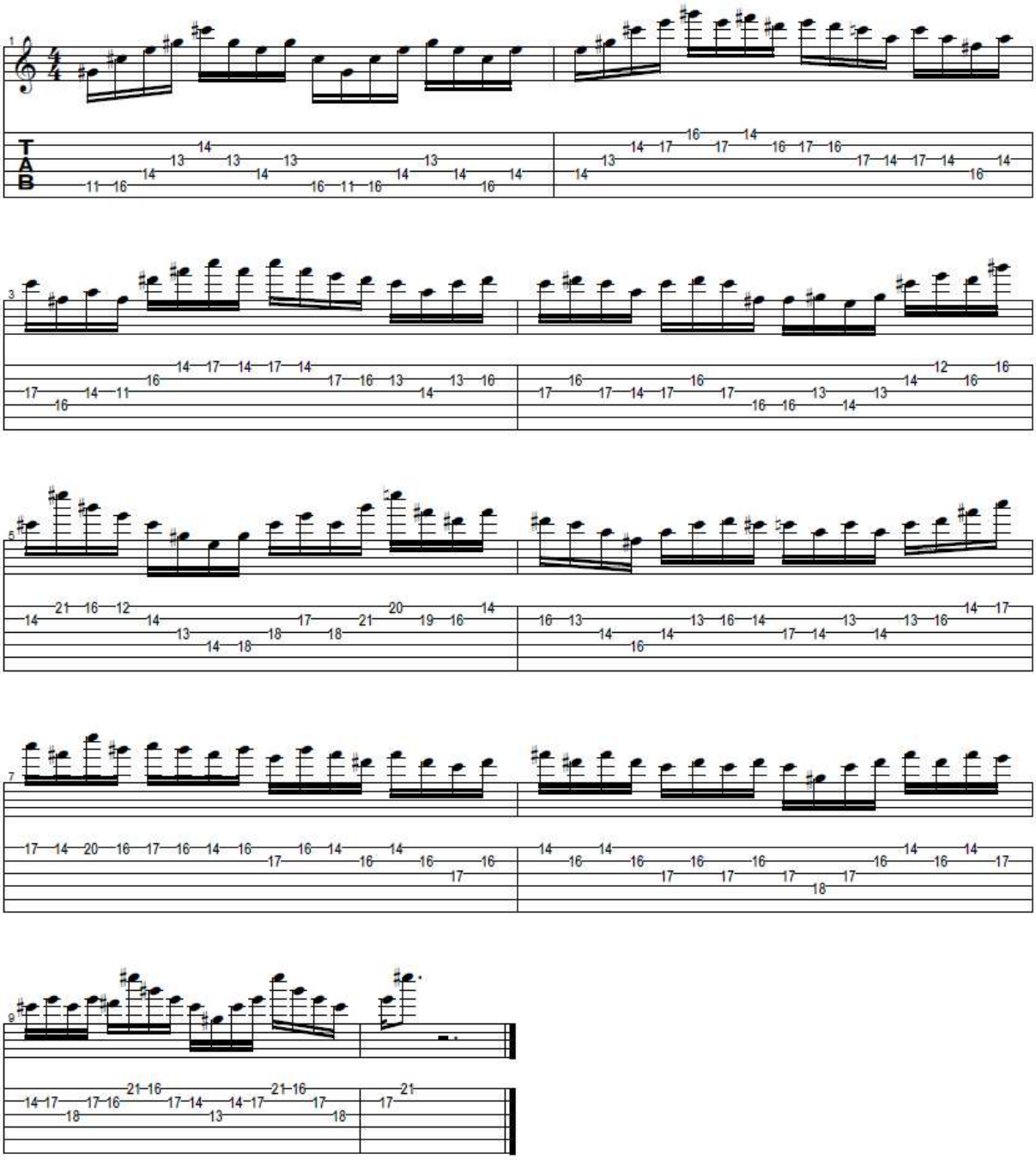
Figuur 27: enkele voorbeelden van de evolutie van de doelfunctiewaarde voor *Algo3*.

Bedenkingen bij Algo3

Afgaande op de evolutie van de doelfunctiewaarde, produceert *Algo3* ongeveer gelijkwaardige oplossingen als *Algo2v2CEM2*. Het gevonden optimum kent eveneens een reductie tot 30% à 40% van de doelfunctiewaarde ten opzichte van de eerste oplossing. *Algo3* komt echter in heel wat minder tijd tot deze reductie. Subjectief gezien klinken de gegenereerde stukken van *Algo3* ook een stuk beter. Waar de muziek gemaakt door *Algo2v2CEM2* nog voor een deel willekeurig of zelfs wat raar klonk, is dit voor *Algo3* nauwelijks nog het geval. Door groepen van noten te veranderen met korte Markov ketens, klinkt de muzikaliteit van de referentiesolo heel wat beter door in de gegenereerde oplossingen. Hier en daar zijn er zelfs elementen van de referentiesolo rechtstreeks terug te vinden in de geproduceerde muziek. Ook qua stijlkenmerken kan men bijvoorbeeld in maten 5 en 6 van de solo uit figuur 29 een aantal arpeggio's terugvinden. In maten 7 en 8 is er een orgelpunt te ontwaren. De bedoeling van het project *soloGen* was om aanhoorbare tot zelfs goede variaties op een enkele solo te creëren, waarvoor bovendien een aantal neoklassieke stijlkenmerken van de betreffende solo in terug te vinden zijn. Met *Algo3* is dit gelukt.



Figuur 28: een solo gegenereerd door *Algo3* met hier en daar een chaotische sprong.



Figuur 29: een solo gegenereerd door *Algo3* met aangepaste perturbatieregel, zonder chaotische sprongen.

7.3.2 Het project ants

In het project *ants* worden er solo's gegenereerd op basis van een referentiesolo, aan de hand van algoritmes die gebaseerd zijn op Ant Colony Optimization. ACO werd reeds besproken in deel 4. De Java code van het project ants is in zijn geheel terug te vinden in bijlage VI.

7.3.2.1 De doelfunctie

De doelfunctie in het project *ants* is anders dan in voorgaande delen. De term die aansluiting zoekt bij de jSymbolic referentiestatistieken blijft bewaard. Daar wordt een term aan toegevoegd die de samenstelling van noten in de oplossing beïnvloedt. De nieuwe doelfunctie wordt weergegeven in formule (7.3).

$$F(s) = \min\left\{\sum_{i=1}^{21}\frac{x_i - y_i}{y_i} + \sum_{i=1}^{l} w * (n_i - r_i)^2\right\} \tag{7.3}$$

Een referentiesolo bestaat uit l unieke elementen, de solo uit figuur 5 heeft er zo 19. Elke unieke noot i komt r_i keer voor in deze referentiesolo en n_i keer voor in de gegenereerde oplossing. Het doel is om in de algoritmisch gegenereerde solo ongeveer dezelfde samenstelling van noten te bekomen als in de referentiesolo. Om dit te bereiken wordt in de doelfunctie het gekwadrateerde verschil van n_i en r_i genomen. Bijgevolg zal een sterk afwijkende samenstelling van de oplossing ten opzichte van de referentiesolo zwaar worden bestraft. De parameter w is een vaste coëfficiënt die ervoor moet zorgen dat het tweede deel van de doelfunctie ongeveer van dezelfde grootteorde is als het eerste deel. Deze parameter wordt vastgelegd op w=0,1.

De bedoeling van deze nieuwe toevoeging is om ervoor te zorgen dat noten die gemakkelijk worden gekozen niet in overdaad voorkomen in de oplossing, en vice versa dat noten die niet gemakkelijk gekozen worden toch meer kans maken om voor te komen. De reden waarom dit zou werken wordt duidelijk in wat volgt.

7.3.2.2 Aanpassing van de Ant System heuristiek

In deel 4.1.2 werd uitgelegd hoe de Ant System heuristiek werkt met bijhorende formules. Deze formules moeten aangepast worden om te kunnen gebruiken in het project *ants*.

Om te beginnen is er de random proportional transition rule. In deze formule komt de onafhankelijke variabele *zichtbaarheid* of η_{ij} voor, die de inverse is van de afstand die een 'mier' k moet afleggen tussen twee punten i en j. In de context van dit project geven we er een eigen betekenis aan. De variabele wordt hier ingevuld door de Markov transitiekans p_{ij} om van noot i naar noot j te gaan. Naar analogie met Ant System betekent dit dat hoe groter de transitiekans is, hoe kleiner de metaforische afstand is die een mier tussen de twee noten moet afleggen. De random proportional transition rule voor dit project wordt dan:

$$P_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha} \cdot [p_{ij}]^{\beta}}{\sum_{l}[\tau_{il}(t)]^{\alpha} \cdot [p_{il}]^{\beta}} \tag{7.4}$$

We moeten ook vastleggen hoe de 'feromonensporen' tot stand komen. Dit staat in functie van de totale afstand die een mier heeft afgelegd op zijn pad $T^k(t)$ tijdens iteratie t. Deze figuurlijke totale afstand wordt in het project *ants* gemeten via de doelfunctiewaarde $F(s_k)$ van de oplossing die de mier heeft gemaakt. De grootte van het feromonenspoor τ_{ij} dat mier k achterlaat wordt dan:

$$\Delta\tau_{ij}^{k}(t) = \frac{1}{F(s_k)} als\ (i,j)\ \epsilon\ T^{k}(t), anders\ 0. \qquad (7.5)$$

Een mier die een goede oplossing creëert, zal dus een groot spoor mogen achterlaten. De *intensiteit* variabele van de random proportional transition formule wordt bijgevolg beïnvloed door de doelfunctie. Transities die voorkomen in goede oplossingen worden dus geprefereerd boven transities die voorkomen in slechte oplossingen. Om ervoor te zorgen dat een mier niet telkens terug zou gaan naar die ene transitie met een hoge intensiteit, wordt de intensiteit per mier sterk verminderd indien deze transitie teveel voorkomt in de oplossing $T^k(t)$. Het is dus om die reden dat er een toevoeging is gebeurd aan de doelfunctie. Indien een mier een transitie referentieel gezien teveel kiest, dan wordt het spoor op deze transitie automatisch terug afgebouwd.

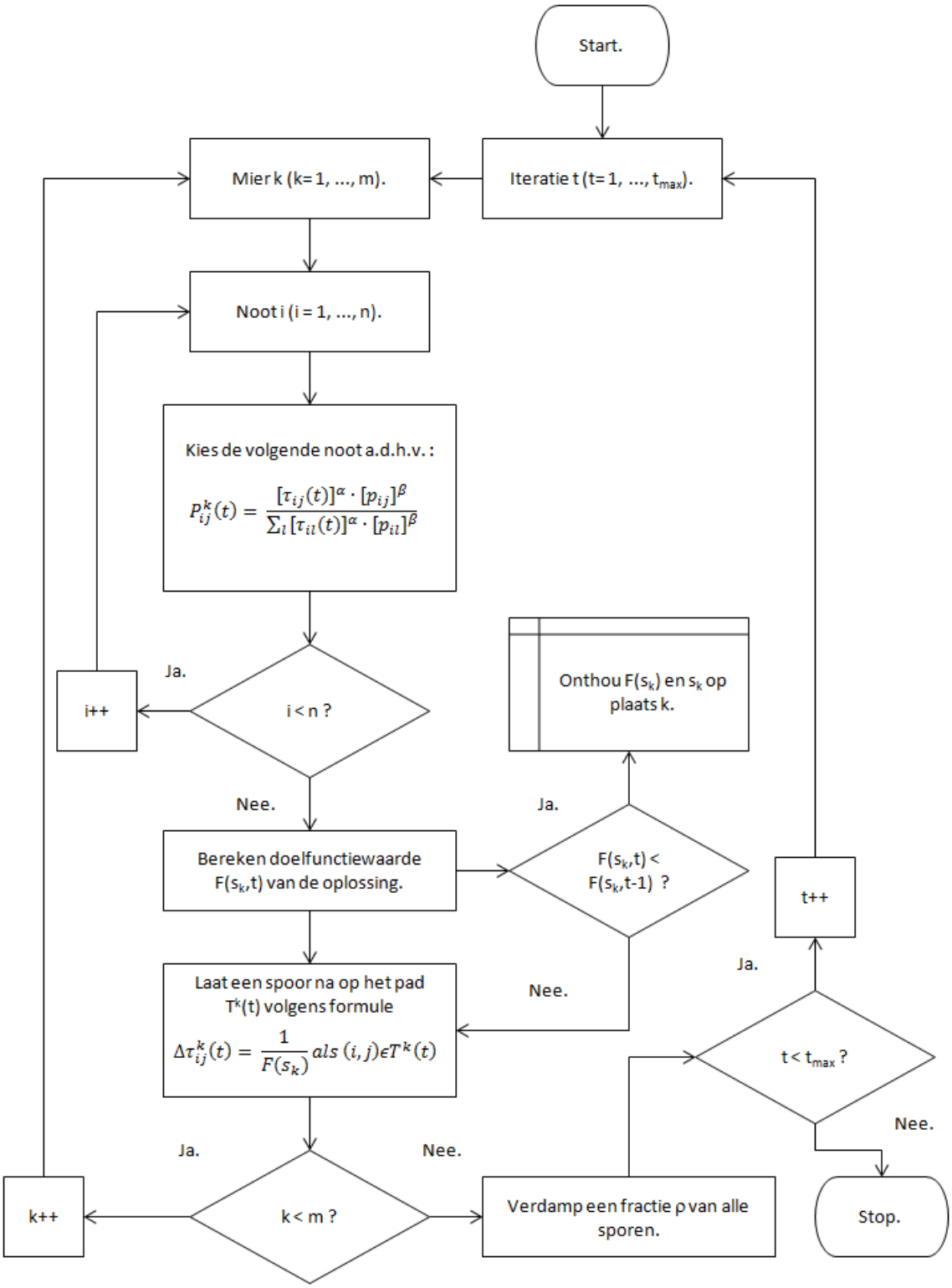
7.3.2.3 Het algoritme

Figuur 30 geeft een process flow chart weer van het algoritme van project *ants*. De Java code van dit algoritme is terug te vinden in bijlage VI.1.

Elke iteratie en vervolgens voor elke mier wordt er een oplossing van *n* noten gebouwd aan de hand van formule (7.4). Merk op dat indien de parameter α=0, mier *k* gewoon een Markov keten maakt. De *intensiteit* variabele valt dan immers weg. Vervolgens valt ook de noemer van de formule weg, aangezien de som van alle p_{ij} voor een gegeven *i* en voor *j* gaande van *j*= 1, ..., *l* met *l* het aantal unieke elementen in de oplossing, altijd gelijk moet zijn aan 1. Dit komt overeen met de sommatie van de elementen van rij *i* in de transitiematrix, waarvan reeds is besproken dat de rijtotalen altijd moeten resulteren in 1 (zie deel 5). Met andere woorden, de random proportional transition regel wordt dan gereduceerd tot p_{ij}, waardoor mier *k* een eenvoudige Markov keten maakt.

Het algoritme, zoals het geïmplementeerd is, kan bekeken worden als *m* parallelle Local Search algoritmes, die elkaar beïnvloeden indien $\alpha \neq 0$. Elke mier maakt een volledige oplossing. Indien deze oplossing beter is dan de vorige oplossing die de mier heeft geproduceerd, dan wordt de oude oplossing vervangen en de nieuwe onthouden. Na afloop van het algoritme wordt de beste oplossing geselecteerd uit een geheugen van *m* oplossingen.

De mieren beïnvloeden elkaar door manipulatie van de random proportional transition rule. Elke iteratie veranderen de berekende kansen, dankzij de 'feromonensporen' en dus via de *intensiteit* variabele in de formule. De grootte van deze feromonensporen staat in functie van de doelfunctiewaarde van de oplossing die een mier maakt. In feite worden oplossingen dus gegenereerd aan de hand van dynamisch gemanipuleerde Markov transitiekansen, die in functie staan van de doelfunctiewaarden en Markov ketens gegenereerd in voorgaande iteraties. Figuur 32 geeft zo een gegenereerde oplossing weer.

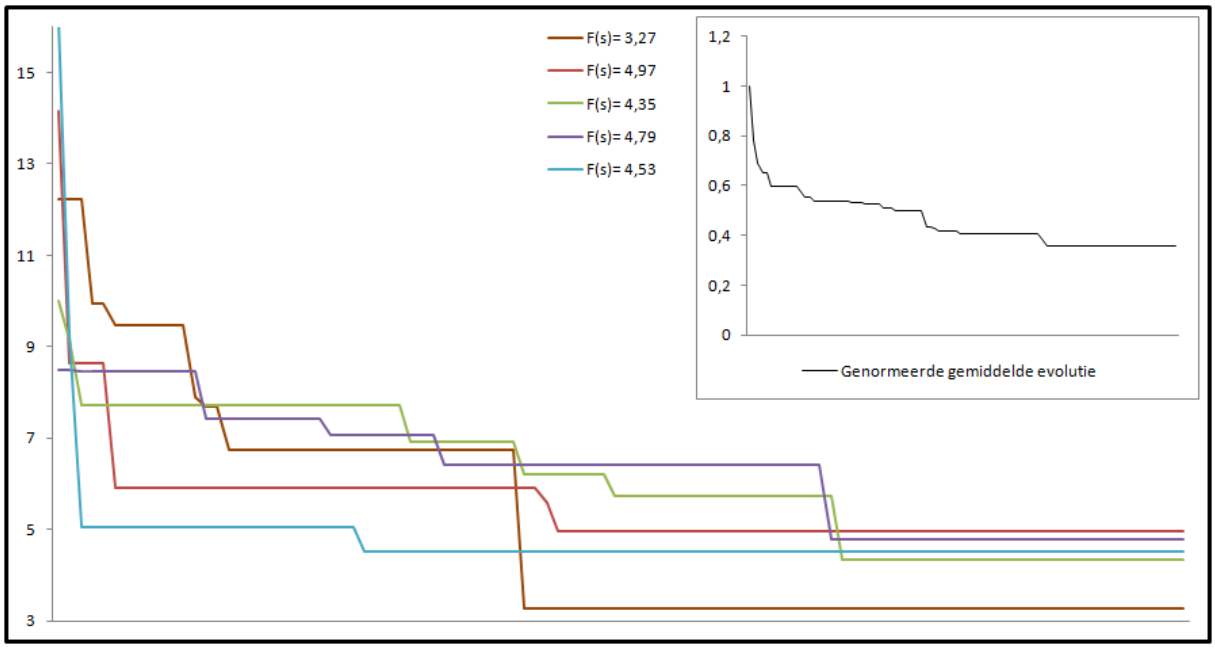


Figuur 30: een process flow chart van het project *ants* algoritme.

Bedenkingen bij ants

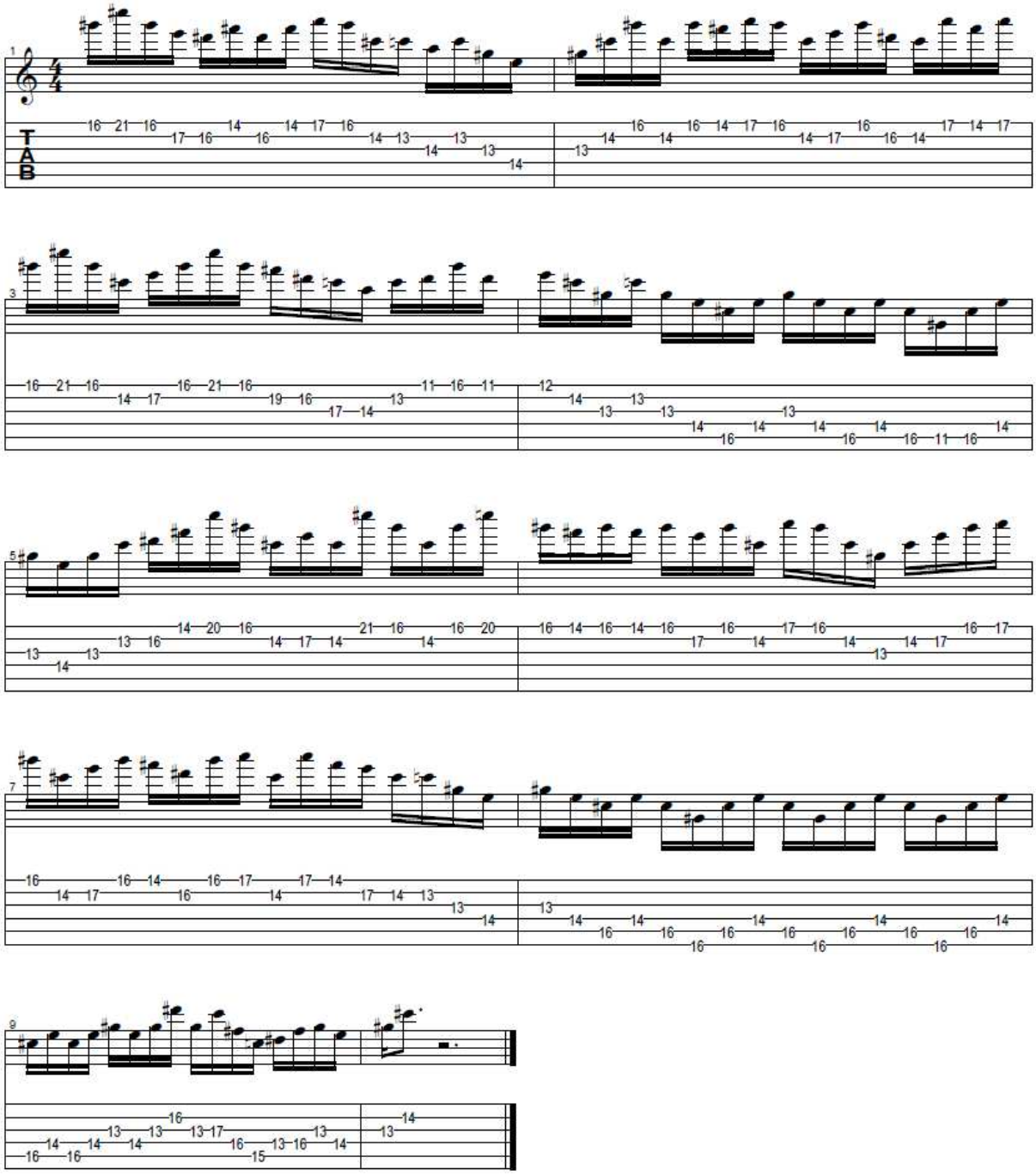
Proefondervindelijk is vastgesteld dat *ants* het beste werkt voor lage waarden van α. Het voorbeeld in figuur 32 is gegenereerd door het algoritme waarbij α=0,1. De intensiteit laag houden en de variabiliteit hoog, gaf subjectief gezien steevast de beste oplossingen. In dat geval komt het er eigenlijk op neer dat de Markov transitiekansen licht, maar effectief gemanipuleerd worden. De verdampingsfactor ρ wordt ook laag gehouden, zo rond ρ=0,005. De τ waarden 'verdampen' dus relatief traag over de iteraties. Wanneer er experimenten gedaan worden met deze waarden voor de parameters, is te zien dat de τ waarden over de iteraties heen naar een bepaalde waarde toe evolueren en daar vervolgens rond blijven schommelen. De doelfunctiewaarden dalen het snelste bij deze vastlegging van de parameters. Een aantal voorbeelden van de evolutie van de 'beste mier' per iteratie, is te zien in figuur 31.

Merk op dat er nog steeds jSymbolic statistieken moeten worden berekend. Zoals eerder besproken vormen deze de achilleshiel in de algoritmes qua executietijd. Voor experimenten met veel mieren levert dit een probleem. Echter, experimenten wezen uit dat een goede oplossing kan bekomen worden met weinig mieren en bovendien met weinig iteraties. De solo uit figuur 32 is gegenereerd met 20 mieren en 100 iteraties. In dat geval heeft het algoritme 45 seconden nodig om zijn taak uit te voeren.



Figuur 31: de evolutie van enkele doelfunctiewaarden, 20 mieren en 100 iteraties.

De muziekstukken gegenereerd met het project *ants* klinken redelijk goed tot goed. Subjectief gezien klinken ze iets beter dan de stukken gemaakt door *Algo3* uit het project *soloGen*, alhoewel dat ze qua aanhoorbaarheid toch ongeveer gelijkwaardig zijn. Ook *ants* produceert de gezochte stijlkenmerken. In maat 8 bijvoorbeeld wordt dezelfde arpeggio drie keer herhaald. Bovendien wordt deze groep van arpeggio's symmetrisch ingesloten door een andere arpeggio. Een dergelijke graad van structuur is een redelijk onverwacht resultaat. In maat 6 kan men ook een orgelpunt terugvinden.



Figuur 32: een solo gegenereerd door het *ants* algoritme.

8 Conclusie

Het doel van deze thesis was om algoritmes te ontwikkelen die neoklassieke gitaarmuziek genereren. Deze gegenereerde muziek moest ten eerste zinvol klinken, en ten tweede onmiskenbaar tot de neoklassieke stijl behoren. Meer specifiek moest de volgende vraag beantwoord worden:

Wat is de beste benaderingswijze en welke zijn de meest efficiënte en effectieve heuristieken voor het algoritmisch genereren van neoklassieke gitaarmuziek?

Na tal van experimenten en vele pagina's geproduceerde Java code, kan eenduidig een antwoord op de vraag geformuleerd worden. De beste gevonden benaderingswijze is de zogenaamde top down- of *shredder*benadering, waarbij een enkele volledige solo als referentie dient. Deze benadering is niet enkel de meest eenvoudige, er moet namelijk niet veel informatie ingevoerd of bewerkt worden, het resulteert bovendien in de beste muziek. Het gebruik van een summiere referentie werpt op zichzelf al een grote restrictie op, namelijk die van de beperkte informatie. Beperkte informatie is in deze meer een deugd dan een probleem op zich. Van die ene referentiesolo worden er tal van statistieken berekend over de melodische aspecten ervan, en worden de kansen voor de overgangen tussen noten geschat. Aan de hand van deze kennis wordt een algoritme opgesteld, dat oplossingen maakt met de elementen van de referentiesolo. De algoritmes die het meest succesvol zijn maken een combinatie tussen het stochastische proces van Markov enerzijds en metaheuristieken anderzijds. Meer bepaald worden oplossingen gegenereerd aan de hand van Markov ketens, en vervolgens worden deze geoptimaliseerd met een zoekalgoritme. Dit zoekalgoritme minimaliseert een doelfunctie, die opgesteld wordt in functie van de statistieken die uit de referentie worden gehaald. Algoritmes die geen gebruik maken van Markov ketens, of algoritmes die enkel Markov gebaseerd zijn en niet optimaliseren, geven subjectief gezien inferieure resultaten tegenover algoritmes die wel beide technieken gebruiken. Er werden twee metaheuristieken gebruikt, verschillende toepassingen van Local Search en een experiment met Ant Colony Optimization. Beide technieken resulteren in gelijkwaardige oplossingen. Het voordeel van Local Search is dat het conceptueel eenvoudiger is en minder parameters heeft, het voordeel van Ant Colony is dat het qua muzikaliteit iets verfijndere oplossingen maakt.

Een opmerkelijk resultaat is wel dat alhoewel er nauwelijks gebruik is gemaakt van compositorische regels, er toch zinvolle muziek kan gegenereerd worden op een quasi volledig stochastische manier. Bovendien gebeurt dit met technieken die in feite uit de economische wetenschappen komen. Buiten hier en daar de beperking op een toonladder, kwam er geen muziektheorie aan te pas.

Een kanttekening moet wel gemaakt worden, dat het probleem 'muziek' sterk gereduceerd werd inzake complexiteit. Aspecten zoals bijvoorbeeld ritme of polyfonie werden volledig genegeerd. Een volgende stap in de lijn van deze thesis zou kunnen zijn om deze aspecten wel in rekening te brengen.

Op het vlak van optimalisatie is het verder zoeken naar een betere optimalisatie voor het behandelde probleem redelijk overbodig. Als we kijken naar de beste algoritmes, dan zien we dat het niet zozeer de reductie is van de doelfunctie is die de muzikaliteit van een gegenereerd stuk bepaald, maar wel de aard van het algoritme. De algoritmes die de meest aanhoorbare muziek maken, zijn niet per se de algoritmes die de laagste doelfunctiewaarde leveren. Hetgeen volgens de auteur nog wel een verschil zou kunnen maken, is het Local Search aspect van ACO aan te passen en iets te verfijnen.

Een opmerking die gemaakt kan worden bij de gecreëerde muziek, is dat deze hoe dan ook redelijk themaloos blijft. Dit ligt enerzijds aan het feit dat er solo's worden gegenereerd. Solo's zijn sowieso thematisch minder sterk dan bijvoorbeeld refreinen. Anderzijds kan dit verklaard worden door dat de methodes die gebruikt worden niet naar patronen zoeken en grotendeels geheugenloos zijn. Een aanbeveling voor verdere ontwikkeling in de lijn van deze thesis, is dan ook om meer naar patronen in muziek te zoeken en methodes te integreren die wel meer tijdsvariant zijn dan de gebruikte methodes.

Bibliografie

1. Wikipedia (2014). *Neoclassical metal*. Retrieved from http://en.wikipedia.org/wiki/Neoclassical_metal

2. Monk, L. (2012). *Yngwie Malmsteen: less is more... no more is more*. Retrieved from http://www.truthinshredding.com/2012/09/yngwie-malmsteen-less-is-more-no-more.html

3. Marano, S. (2013). *Put the Pedal to the Neo-Classical Metal*. Retrieved from http://www.guitarworld.com/guitar-strength-put-pedal-neo-classical-metal

4. Pilhofer, M., Day, H., (2009). *Muziektheorie voor Dummies*. Amsterdam: Pearson Education.

5. Ultimate Guitar (2007). *Neo-Classical Metal Music*. Retrieved from http://www.ultimate-guitar.com/lessons/music_styles/neo-classical_metal_music.html

6. Wikipedia (2014). *Arpeggio*. Retrieved from http://nl.wikipedia.org/wiki/Arpeggio

7. Wikipedia (2013). *Pedal point*. Retrieved from http://en.wikipedia.org/wiki/Pedal_point

8. Mosteller, F., Youtz, C., (1968). *Mozart's Random Music*. Retrieved from www.stat.harvard.edu

9. Jacob, B., (1996). Algorithmic composition as a model of creativity. *Organized Sound*, 1, 157-165. doi: 10.1017/S1355771896000222

10. Edwards., M., (2011). Algorithmic Composition: Computational Thinking in Music. *Communications of the ACM*, 54(7), 58-67. doi: 10.1145/1965724.1965742

11. Assayag, G., (1998). *Computer Assisted Composition Today*. Retrieved from: http://repmus.ircam.fr/

12. Herremans, D., Sörensen, K., (2012). *Composing Fifth Species Counterpoint Music With Variable Neighbourhood Search*. Retrieved from http://antor.ua.ac.be/publication/composing-fifth-species-counterpoint-music-variable-neighborhood-search

13. Pachet, F., Roy, P. (2010). *Markov constraints: steerable generation of Markov sequences*, 10-11. doi: 10.1007/s10601-010-9101-4

14. Bäckman, K. (2010). *Ant Colony Optimization and Evolutionary Algorithms Applied to Jazz solo Improvisation*. Retrieved from http://cumulus.ei.hv.se/~imkb/kjell/eji/eji.htm.

15. Geis, M., Middendorf, M. (2008). Creating melodies and baroque harmonies with ant colony optimization. *International Journal of Intelligent Computing and Cybernetics*, Vol. 1 Issue 2, 213-238.

16. De Wieuw, F., Gijsbrechts, J., Hein, M. (2012). *Optimalisatietechnieken op basis van insecten*. (eigen bachelorproef)

17. Parys,W., & Pauwels, W., (2010). *Optimalisering*. Antwerpen: Universitas

18. Springael, J., (2013). *Operationeel Onderzoek*. Blackboard, Universiteit Antwerpen.

19. Sörensen, K., & Glover, F. (n.d.). *Metaheuristics*. Retrieved from *www.opttek.com/sites/default/files/Metaheuristics.pdf*

20. Pinedo, M.,(n.d.). *Heuristic Algorithms: Local Search.* Retrieved from http://www.stern.nyu.edu/om/faculty/pinedo/scheduling/shakhlevich/handout10.pdf

21. Mladenovic, N., & Hansen, P., (1997). *Variable Neighbourhood Search*. doi: 10.1016/S0305-0548(97)00031-2

22. Dorigo, M., Stützle, T., (2004). *Ant Colony Optimization*. Massachusetts: MIT Press.

23. Dréo, J., Pétrowski, A., Siarry, P., & Taillard, E., (2006). *Metaheuristics for Hard Optimization*. Berlijn: Springer.

24. Glover, F. (1990). Tabu Search: A Tutorial. *INTERFACES*, 20, 74-94.

25. Anderson, D. (2011). *Discrete Time Markov Chains*. Retrieved from http://www.math.wisc.edu/~anderson/605F11/Notes/StochBioChapter3.pdf

26. Grinstead, C. M., Snell, J. L., (1997). *Introduction to Probability*. Retrieved from http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html

27. Jones, M. T., (2005*). Estimating Markov Transition Matrices Using Proportions Data: An Application to Credit Risk*. Retrieved from https://www.imf.org/external/pubs/ft/wp/2005/wp05219.pdf

28. Hansen, B., (n.d.). *Musical Instrument Digital Interface*. Retrieved from www.hansenb.pdx.edu/pdf/MIDI.pdf

29. Wolfe, J., (n.d.). *Note names, MIDI numbers and frequencies*. Retrieved from http://www.phys.unsw.edu.au/jw/notes.html

30. Downie, J. S. (2003). Music information retrieval. *Annual review of information science and technology*, *37*(1), 295-340.

31. McKay, C. (2010). *Automatic music classification with jMIR,* p. 171-238 (Doctoral dissertation, McGill University).

32. MakeMusic (2011). *Middle C (MIDI note 60) is C3/C4/C5?*. Retrieved from http://forum.makemusic.com/default.aspx?f=5&m=327431

33. Moore, D., McGabe, G., Craig, B., (2007). Chapter 15: Nonparametric Tests. *Introduction to the Practice of Statistics*. Retrieved from http://bcs.whfreeman.com/ips6e/content/cat_040/pdf/ips6e_chapter15.pdf.

34. Hole, G., (2011). *Graham Hole's Research Skills: The Wilcoxon Test*. Retrieved from http://www.sussex.ac.uk/Users/grahamh/RM1web/WilcoxonHandoout2011.pdf

35. Zaiontz, C., (2014). *Wilcoxon Signed-Ranks Table*. Retrieved from http://www.real-statistics.com/statistics-tables/wilcoxon-signed-ranks-table/

36. Wikipedia (2014). *Cross Entropy*. Retrieved from http://en.wikipedia.org/wiki/Cross_entropy

37. Cogill, R., (2011). *Introduction to Markov Chains*. Retrieved from http://people.virginia.edu/~rlc9s/sys6005/SYS_6005_Intro_to_MC.pdf

Bronnen van de afbeeldingen

[Fig. 3] Ultimate Guitar (2006). *Yngwie Malmsteen - Arpeggio's From Hell GP4 file* [software]. Retrieved from http://tabs.ultimateguitar.com/y/yngwie_malmsteen/arpeggios_from_hell_guitar_pro.htm

[Fig. 5] Ultimate Guitar (2011). *Dream Theater - Build Me Up, Break Me Down GP5 file* [software]. Retrieved from http://tabs.ultimateguitar.com/d/dream_theater/build_me_up_break_me_down_ver2_guitar_pro.htm

Bijlagen

Nota bij de bijlagen

In de bijlagen zit alle geproduceerde Java code. Allereerst is er de class MidiMaker terug te vinden, hetgeen in elke package opnieuw voorkomt. Vervolgens is het project *markovchainer* terug te vinden. Dan worden alle zelf geschreven statische Java methods van de jSymbolic statistieken weergegeven. Deze methods horen bij zowel bij *pedalpoints*, *soloGen* als *ants*. De jSymbolic statistieken worden apart weergegeven, omdat het geen zin heeft om ze drie keer opnieuw in de bijlage te steken. Na de jSymbolic statistieken volgen de gerelateerde projecten.

Alle code is zelf geschreven met uitzondering van de class MidiMaker. Deze class maakt een MIDI file van de output van de algoritmes, met behulp van javax.sound.midi.*. Deze class komt in elk project terug voor en komt van de volgende bron:

Brown, K., (2003). *How to write a Java program that writes a MIDI file*. Retrieved from http://www.automatic-pilot.com/midifile.html

De class is wel aangepast om de kunnen gebruiken in de context van de projecten. Classes met de jSymbolic statistieken zijn zelf geschreven, maar uiteraard opgesteld aan de hand van de originele files. jSymbolic is geschreven door Cory McKay.

Verder is het nog nuttig om te vermelden in welke classes de main() functies verstopt zitten:

markovchainer → *MarkovMain*

pedalpoints → *PedalPointer*

soloGen → *SoloGenMain*

ants → *AntSystemMain*

Bijlage I: MidiMaker

```
/**
 * original MidiFile by Karl Brown
 *
 * Brown, K., (2003). How to write a Java program that writes a MIDI
 * file. Retrieved from http://www.automatic-pilot.com/midifile.html
 *
 * adaptation:
 *
 * @author Max
 *
 */

import java.io.File;
import java.util.Vector;

import javax.sound.midi.MetaMessage;
import javax.sound.midi.MidiEvent;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Sequence;
import javax.sound.midi.ShortMessage;
import javax.sound.midi.SysexMessage;
import javax.sound.midi.Track;

public class MidiMaker{

    public static void writeMidiFile(Vector<Integer> NoteVector){

        System.out.println("midifile begin ");

        try{
            // **** Create a new MIDI sequence with 32 ticks per beat
            // ****
            Sequence s=
            new Sequence(javax.sound.midi.Sequence.PPQ, 38);

            // **** Obtain a MIDI track from the sequence ****
            Track t= s.createTrack();

            // **** General MIDI sysex -- turn on General MIDI sound
            // set ****
            byte[] b=
            {(byte) 0xF0, 0x7E, 0x7F, 0x09, 0x01, (byte) 0xF7};
            SysexMessage sm= new SysexMessage();
            sm.setMessage(b, 6);
            MidiEvent me= new MidiEvent(sm, (long) 0);
            t.add(me);

            // **** set tempo (meta event) ****
            MetaMessage mt= new MetaMessage();
            byte[] bt= {0x02, (byte) 0x00, 0x00};
            mt.setMessage(0x51, bt, 3);
            me= new MidiEvent(mt, (long) 0);
            t.add(me);

            // **** set track name (meta event) ****
            mt= new MetaMessage();
            String TrackName= new String("midifile track");
            mt.setMessage(0x03, TrackName.getBytes(),
```

```
            TrackName.length());
            me= new MidiEvent(mt, (long) 0);
            t.add(me);

            // **** set omni on ****
            ShortMessage mm= new ShortMessage();
            mm.setMessage(0xB0, 0x7D, 0x00);
            me= new MidiEvent(mm, (long) 0);
            t.add(me);

            // **** set poly on ****
            mm= new ShortMessage();
            mm.setMessage(0xB0, 0x7F, 0x00);
            me= new MidiEvent(mm, (long) 0);
            t.add(me);

            // **** set instrument to Piano ****
            mm= new ShortMessage();
            mm.setMessage(0xC0, 0x1A, 0x00);
            me= new MidiEvent(mm, (long) 0);
            t.add(me);

            // loop
            //
************************************************************************

            // maken van een loop die de vector afspeelt in midi

            int lengte= NoteVector.size();
            int ticks= 40;

            for(int k= 0; k< lengte- 1; k++){

                // **** note on ****
                mm= new ShortMessage();
                mm.setMessage(0x90, NoteVector.elementAt(k), 0x60);
                me= new MidiEvent(mm, (long) 1+ (k)* ticks);
                t.add(me);

                // **** note off ticks ticks later ****
                mm= new ShortMessage();
                mm.setMessage(0x80, NoteVector.elementAt(k), 0x40);
                me= new MidiEvent(mm, (long) ticks+ (k)* ticks);
                t.add(me);

            }

            // **** note on ****
            mm= new ShortMessage();
            mm.setMessage(0x90,
            NoteVector.elementAt(NoteVector.size()- 1), 0x60);
            me= new MidiEvent(mm, (long) 1+ (lengte- 1)* ticks);
            t.add(me);

            // **** note off ticks ticks later ****
            mm= new ShortMessage();
            mm.setMessage(0x80,
            NoteVector.elementAt(NoteVector.size()- 1), 0x40);
            me= new MidiEvent(mm, (long) 4* ticks+ (lengte)* ticks);
```

```
            t.add(me);

            //
**********************************************************************

            // **** set end of track (meta event) 140 ticks later ****
            mt= new MetaMessage();
            byte[] bet= {}; // empty array
            mt.setMessage(0x2F, bet, 0);
            me= new MidiEvent(mt, (long) 140);
            t.add(me);

            // **** write the MIDI sequence to a MIDI file ****
            File f= new File("tester.mid");
            MidiSystem.write(s, 1, f);
        } // try
        catch(Exception e){
            System.out.println("Exception caught "+ e.toString());
        } // catch
        System.out.println("midifile end ");
    }

}
```

Bijlage II: markovchainer

Bijlage II.1: MarkovChainer

```
package markovchainer;

import java.util.HashMap;
import java.util.Vector;

/**
 * Maakt een Markov Chain gegeven een transitiematrix.
 *
 * @author Max
 *
 */

public class MarkovChainer{

    public static Vector<Integer> makeChain(int aantalnoten,
    int firstnote, Vector<Integer> MidiElements,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans){

        Vector<Integer> MarkovChain= new Vector<Integer>(aantalnoten);
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();
        double P= 0;
        double rand= 0;
        int nieuwelement= 0;

        for(int i= 0; i< aantalnoten; i++){
            if(i== 0){
                MarkovChain.add(firstnote);
            }else{
                do{
                    rand= Math.random();
                }while(rand< 0.001);

                P= 0;
                for(int j= 0; j< MidiElements.size()+ 1; j++){

                    if(P< rand){

                        InnerMap=
                        NotesMarkovTrans.get(MarkovChain
                        .elementAt(i- 1));
                        P+= InnerMap.get(MidiElements.elementAt(j));
                    }else{
                        nieuwelement= MidiElements.elementAt(j- 1);
                        break;
                    }
                }
                MarkovChain.add(nieuwelement);
            }
        }

        return MarkovChain;
    }
}
```

Bijlage II.2: MarkovMain

```
package markovchainer;

import java.util.Collections;
import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Main class van markovchainer
 *
 * @author Max
 *
 */

public class MarkovMain{

    public MarkovMain(){

        int[] MidiSeq=
        new int[]{65, 67, 68, 71, 72, 74, 75, 74, 72, 71, 72, 71, 68,
        71, 68, 67, 65, 67, 68, 71, 72, 74, 75, 72, 67, 72, 74, 72,
        71, 72};

        int aantalnoten= MidiSeq.length;

        Vector<Integer> MidiSequentie= new Vector<Integer>(0);
        Vector<Integer> MidiElements= new Vector<Integer>(0);

        for(int i= 0; i< MidiSeq.length; i++){
            MidiSequentie.add(MidiSeq[i]);
        }

        // Een vector vullen met de unieke elementen uit MidiSeq
        for(int i= 0; i< MidiSeq.length; i++){
            if(MidiElements.contains(MidiSeq[i])){}else{
                MidiElements.add(MidiSeq[i]);
            }
        }
        System.out.println("MidiElements: "+ MidiElements);

        Collections.sort(MidiElements);

        // 1ste orde transitiematrix van de noten
        HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();
        NotesMarkovTrans= NotesMarkovizer.makeMarkov(MidiSequentie);
        System.out.println("NotesMarkovTrans: "+ NotesMarkovTrans);

        // Maak een oplossing gebaseerd op Markov 1ste orde TM (noten,
        // niet intervallen)
        int firstnote= 0;
        int r= MidiElements.size();
        Vector<Integer> Oplossing= new Vector<Integer>(aantalnoten);
        Random random= new Random();
        int index= 0;

        index= random.nextInt(r);
        firstnote= MidiElements.elementAt(index);
```

```
        Oplossing=
        MarkovChainer.makeChain(aantalnoten, firstnote, MidiElements,
        NotesMarkovTrans);

        // Oplossing schrijven
        MidiMaker.writeMidiFile(Oplossing);
        System.out.println("Oplossing: "+ Oplossing);
    }

    // Main
    public static void main(String[] args){
        new MarkovMain();
    }
}
```

Bijlage II.3: NotesMarkovizer

```
package markovchainer;

import java.util.Collections;
import java.util.HashMap;
import java.util.Vector;

/**
 * Deze class maakt een Markov Transitiematrix van een reeks gegeven
 * midinummers in Main.
 *
 * @author Max
 *
 */

public class NotesMarkovizer{

    public static HashMap<Integer,HashMap<Integer,Double>>
    makeMarkov(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();

        Vector<Integer> MidiElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke noten
        for(int i= 0; i< MidiSequentie.size(); i++){
            if(MidiElements.contains(MidiSequentie.elementAt(i))){}else{
                MidiElements.add(MidiSequentie.elementAt(i));
            }
        }

        Collections.sort(MidiElements);

        // Nootparen worden geteld en in een geneste hashmap gestoken
        int counter= 0;
        int jcounter= 0;
        HashMap<Integer,Integer> Rijtotaal=
        new HashMap<Integer,Integer>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){
                for(int k= 0; k< MidiSequentie.size()- 1; k++){
                    if(MidiElements.elementAt(i)== MidiSequentie
                    .elementAt(k)&&
                    MidiElements.elementAt(j)== MidiSequentie
                    .elementAt(k+ 1)){
                        counter++;
                    }
                }// k

                MarkovTrans2.put(MidiElements.elementAt(j),
                (double) counter);
                jcounter+= counter;
                counter= 0;
```

```
            }// j

            MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);

            if(jcounter== 0){
                Rijtotaal.put(MidiElements.elementAt(i), 1);
            }else{
                Rijtotaal.put(MidiElements.elementAt(i), jcounter);
            }// hashmap van rijtotalen om transitiematrix te kunnen
             // maken

            jcounter= 0;
        }// i

        // De hashmap met getelde nootparen wordt per rij gedeeld door
        // de respectievelijke rijtotalen
        // teneinde een transitiematrix te bekomen
        Double absel;
        double relel= 0;
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){

                InnerMap= MarkovTrans.get(MidiElements.elementAt(i));
                absel= InnerMap.get(MidiElements.elementAt(j));
                relel=
                absel/ Rijtotaal.get(MidiElements.elementAt(i));
                MarkovTrans2.put(MidiElements.elementAt(j), relel);
            }

            MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);
        }

        return MarkovTrans;
    }// makeMarkov

}
```

Bijlage III: De jSymbolic methods

Bijlage III.1: AmountOfArpeggiation

```
import java.util.*;

/**
 * Berekent intervallen met grootte 0,3,4,7,10,11,12,15,16.
 *
 * @author Max
 *
 */

public class AmountOfArpeggiation{

    public static double getAmountOfArpeggiation(
    Vector<Integer> Intervals){
        int aoa= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 0||
            Math.abs(Intervals.elementAt(i))== 3||
            Math.abs(Intervals.elementAt(i))== 4||
            Math.abs(Intervals.elementAt(i))== 7||
            Math.abs(Intervals.elementAt(i))== 10||
            Math.abs(Intervals.elementAt(i))== 11||
            Math.abs(Intervals.elementAt(i))== 12||
            Math.abs(Intervals.elementAt(i))== 15||
            Math.abs(Intervals.elementAt(i))== 16){
                aoa++;
            }
        }

        double aoa2= (double) aoa/ Intervals.size();
        return aoa2;
    }
}
```

Bijlage III.2: AverageMelodicInterval

```
import java.util.*;

/**
 * Berekent het gemiddelde interval
 *
 * @author Max
 *
 */

public class AverageMelodicInterval{

    public static double getAverageMelodicInterval(
    Vector<Integer> Intervals){

        int somIntervals= 0;

        for(int i= 0; i< Intervals.size(); i++){
            somIntervals+= Math.abs(Intervals.elementAt(i));
        }

        double ami= (double) somIntervals/ Intervals.size();
        return ami;
    }
}
```

Bijlage III.3: ChromaticMotion

```
import java.util.Vector;

/**
 * Berekent het relatieve aantal intervallen van 1 stap
 *
 * @author Max
 *
 */

public class ChromaticMotion{

    public static double
    getChromaticMotion(Vector<Integer> Intervals){

        int chrome= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 1){
                chrome++;
            }
        }

        double chrome2= (double) chrome/ Intervals.size();
        return chrome2;
    }
}
```

Bijlage III.4: DirectionOfMotion

```
import java.util.*;

/**
 * berekent stijgingen ten opzichte van de totale bewegingen
 *
 * @author Max
 *
 */

public class DirectionOfMotion{

    public static double getDirectionOfMotion(
    Vector<Integer> Intervals){

        int ups= 0;
        int downs= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Intervals.elementAt(i)> 0){
                ups++;
            }else if(Intervals.elementAt(i)< 0){
                downs++;
            }
        }
        double dom= (double) ups/ (ups+ downs);
        return dom;
    }
}
```

Bijlage III.5: DistanceBetweenMostCommonMelodicIntervals

```
import java.util.*;

/**
 * berekent het absolute verschil tussen de 2 meest voorkomende
 * intervallen
 *
 * @author Max
 *
 */

public class DistanceBetweenMostCommonMelodicIntervals{
    public static double
    getDistanceBetweenMostCommonMelodicIntervals(
    Vector<Integer> Intervals){

        int max_int= 0;
        int second_max_int= 0;
        int count= 0;
        int tempcount= 0;
        int count2= 0;
        int tempcount2= 0;

        // Berekenen meest voorkomende interval max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))){
                    tempcount++;
                }
            }
            if(tempcount> count){
                max_int= Math.abs(Intervals.elementAt(i));
                count= tempcount;
            }
            tempcount= 0;
        }
        // Berekenen 2de meest voorkomende interval second_max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))&&
                Math.abs(Intervals.elementAt(i))!= max_int){
                    tempcount2++;
                }
            }
            if(tempcount2> count2){
                second_max_int= Math.abs(Intervals.elementAt(i));
                count2= tempcount2;
            }
            tempcount2= 0;
        }
        // berekenen absolute verschil
        double dbmcmi= Math.abs(max_int- second_max_int);
        return dbmcmi;
    }
}
```

Bijlage III.6: DurationOfMelodicArcs

```
import java.util.*;

/**
 * berekent het gemiddeld aantal noten tussen melodische pieken en
 * dalen
 *
 * @author Max
 *
 */
public class DurationOfMelodicArcs{

    public static double getDurationOfMelodicArcs(
    Vector<Integer> PedalPointVector, Vector<Integer> Intervals){

        int notes_between= PedalPointVector.size();
        int number_arcs= 1;
        double direction= Math.signum(Intervals.elementAt(0));

        // Bereken aantal arcs
        if(Intervals.elementAt(0)== 0){
            for(int i= 0; i< Intervals.size(); i++){
                direction= Math.signum(Intervals.elementAt(i));
                if(direction!= 0){
                    break;
                }
            }
        }

        for(int i= 0; i< Intervals.size()- 1; i++){
            if(Math.signum(Intervals.elementAt(i+ 1))!= 0&&
            direction!= Math.signum(Intervals.elementAt(i+ 1))){
                number_arcs++;
                direction= Math.signum(Intervals.elementAt(i+ 1));
            }
        }

        // Duration of Melodic Arcs berekenen
        double dma= (double) notes_between/ number_arcs;
        return dma;
    }
}
```

Bijlage III.7: IntervalBetweenStrongestPitches

```
import java.util.*;

/**
 * berekent het interval tussen de 2 meest voorkomende frequenties
 *
 * @author Max
 *
 */

public class IntervalBetweenStrongestPitches{
    public static double getIntervalBetweenStrongestPitches(
    Vector<Integer> PedalPointVector){
        int common_pitch= 0;
        int second_common_pitch= 0;
        int count3= 0;
        int tempcount3= 0;
        int count4= 0;
        int tempcount4= 0;
        // Berekenen meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))){
                    tempcount3++;
                }
            }
            if(tempcount3> count3){
                common_pitch= Math.abs(PedalPointVector.elementAt(i));
                count3= tempcount3;
            }
            tempcount3= 0;
        }
        // Berekenen 2de meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))&&
                Math.abs(PedalPointVector.elementAt(i))!= common_pitch){
                    tempcount4++;
                }
            }
            if(tempcount4> count4){
                second_common_pitch=
                Math.abs(PedalPointVector.elementAt(i));
                count4= tempcount4;
            }
            tempcount4= 0;
        }
        // berekenen absolute verschil
        double ibsp= Math.abs(common_pitch- second_common_pitch);
        return ibsp;

    }

}
```

Bijlage III.8: MelodicFifths

```
import java.util.*;

/**
 * berekent het relatieve aandeel van intervallen van 7 stappen
 *
 * @author Max
 *
 */

public class MelodicFifths{

    public static double getMelodicFifths(Vector<Integer> Intervals){

        int fifths= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 7){
                fifths++;
            }
        }

        double frac_fifths= (double) fifths/ Intervals.size();
        return frac_fifths;
    }

}
```

Bijlage III.9: MelodicOctaves

```
import java.util.*;

/**
 * berekent het fractie intervallen met grootte 12
 *
 * @author Max
 *
 */

public class MelodicOctaves{

    public static double getMelodicOctaves(Vector<Integer> Intervals){

        int octaves= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 12){
                octaves++;
            }
        }

        double frac_octave= (double) octaves/ Intervals.size();
        return frac_octave;
    }
}
```

Bijlage III.10: MelodicThirds

```
import java.util.*;

/**
 * berekent het aandeel intervallen met grootte 3 en 4
 *
 * @author Max
 *
 */

public class MelodicThirds{

    public static double getMelodicThirds(Vector<Integer> Intervals){

        int thirds= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 3||
            Math.abs(Intervals.elementAt(i))== 4){
                thirds++;
            }
        }

        double frac_thirds= (double) thirds/ Intervals.size();
        return frac_thirds;
    }

}
```

Bijlage III.11: MelodicTritones

```
import java.util.*;

/**
 * berekent het aandeel intervallen met grootte 6
 *
 * @author Max
 *
 */
public class MelodicTritones{

    public static double
    getMelodicTritones(Vector<Integer> Intervals){

        int tritones= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 6){
                tritones++;
            }
        }

        double frac_tritone= (double) tritones/ Intervals.size();
        return frac_tritone;

    }

}
```

Bijlage III.12: MostCommonMelodicInterval

```
import java.util.*;

/**
 * berekent het meest voorkomende interval
 *
 * @author Max
 *
 */

public class MostCommonMelodicInterval{

    public static double getMostCommonMelodicInterval(
    Vector<Integer> Intervals){

        int max_int= 0;
        int count= 0;
        int tempcount= 0;

        // Berekenen meest voorkomende interval max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))){
                    tempcount++;
                }
            }
            if(tempcount> count){
                max_int= Math.abs(Intervals.elementAt(i));
                count= tempcount;
            }
            tempcount= 0;
        }

        return max_int;
    }

}
```

Bijlage III.13: MostCommonIntervalPrevalence

```
import java.util.*;

/**
 * berekent het aandeel van het meest voorkomende interval
 *
 * @author Max
 *
 */

public class MostCommonMelodicIntervalPrevalence{

    public static double getMostCommonMelodicIntervalPrevalence(
    Vector<Integer> Intervals){

        int max_int= 0;
        int count= 0;
        int tempcount= 0;

        // Berekenen meest voorkomende interval max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))){
                    tempcount++;
                }
            }
            if(tempcount> count){
                max_int= Math.abs(Intervals.elementAt(i));
                count= tempcount;
            }
            tempcount= 0;
        }

        // berekent het aandeel van dat interval ten opzichte van het
        // totale aantal intervallen
        int max_int_occur= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== max_int){
                max_int_occur++;
            }
        }
        double common_int_prev=
        (double) max_int_occur/ Intervals.size();
        return common_int_prev;
    }

}
```

Bijlage III.14: MostCommonPitchPrevalence

```
import java.util.*;

/**
 * berekent het aandeel van de meest voorkomende pitch ten opzichte
 * van alle pitches
 *
 * @author Max
 *
 */

public class MostCommonPitchPrevalence{

    public static double getMostCommonPitchPrevalence(
    Vector<Integer> PedalPointVector){

        int common_pitch= 0;
        int tempcount3= 0;
        int count3= 0;
        int pitch_occur= 0;

        // Berekenen meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))){
                    tempcount3++;
                }
            }
            if(tempcount3> count3){
                common_pitch= Math.abs(PedalPointVector.elementAt(i));
                count3= tempcount3;
            }
            tempcount3= 0;
        }

        // Berekenen prevalence
        for(int i= 0; i< PedalPointVector.size(); i++){
            if(PedalPointVector.elementAt(i)== common_pitch){
                pitch_occur++;
            }
        }
        double common_pitch_prev=
        (double) pitch_occur/ PedalPointVector.size();
        return common_pitch_prev;
    }

}
```

Bijlage III.15: NumberOfCommonMelodicIntervals

```
import java.util.*;

/**
 * berekent het aantal intervallen met een aandeel van minstens 9%
 *
 * @author Max
 *
 */

public class NumberOfCommonMelodicIntervals{

    public static double getNumberOfCommonMelodicIntervals(
    Vector<Integer> Intervals){

        int common_int_counter= 0;
        Vector<Integer> DontCountTwice= new Vector<Integer>(0);
        Vector<Integer> Intervabs= new Vector<Integer>(0);

        for(int i= 0; i< Intervals.size(); i++){
            Intervabs.add(Math.abs(Intervals.elementAt(i)));
        }

        for(int i= 0; i< Intervabs.size(); i++){
            if(DontCountTwice.contains(Intervabs.elementAt(i))){}else{
                int occurrences=
                Collections.frequency(Intervabs,
                Intervabs.elementAt(i));
                double melodint_stat=
                (double) occurrences/ Intervabs.size();
                if(melodint_stat> 0.09){
                    common_int_counter++;
                }
                DontCountTwice.add(Intervabs.elementAt(i));
            }
        }
        return common_int_counter;
    }
}
```

Bijlage III.16: NumberOfCommonPitches

```
import java.util.*;

/**
 * berekent het aantal pitches met een aandeel van minstens 9%
 *
 * @author Max
 *
 */

public class NumberOfCommonPitches{

    public static double getNumberOfCommonPitches(
    Vector<Integer> PedalPointVector){

        int common_pitch_counter= 0;
        Vector<Integer> DontCountTwice2= new Vector<Integer>(0);

        for(int i= 0; i< PedalPointVector.size(); i++){
            if(DontCountTwice2
            .contains(PedalPointVector.elementAt(i))){}else{
                int occurs=
                Collections.frequency(PedalPointVector,
                PedalPointVector.elementAt(i));
                double pitch_stat=
                (double) occurs/ PedalPointVector.size();
                if(pitch_stat> 0.09){
                    common_pitch_counter++;
                }
                DontCountTwice2.add(PedalPointVector.elementAt(i));
            }
        }
        return common_pitch_counter;
    }

}
```

Bijlage III.17: RelativeStrengthOfMostCommonIntervals

```
import java.util.*;

/**
 * aandeel van het 2de meest voorkomende interval / aandeel meest
 * voorkomende interval
 *
 * @author Max
 *
 */

public class RelativeStrengthOfMostCommonIntervals{

    public static double getRelativeStrengthOfMostCommonIntervals(
    Vector<Integer> Intervals){

        int max_int= 0;
        int second_max_int= 0;
        int count= 0;
        int tempcount= 0;
        int count2= 0;
        int tempcount2= 0;

        // Berekenen meest voorkomende interval max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))){
                    tempcount++;
                }
            }
            if(tempcount> count){
                max_int= Math.abs(Intervals.elementAt(i));
                count= tempcount;
            }
            tempcount= 0;
        }

        // Berekenen 2de meest voorkomende interval second_max_int
        for(int i= 0; i< Intervals.size(); i++){
            for(int j= 0; j< Intervals.size(); j++){
                if(i!= j&&
                Math.abs(Intervals.elementAt(i))== Math.abs(Intervals
                .elementAt(j))&&
                Math.abs(Intervals.elementAt(i))!= max_int){
                    tempcount2++;
                }
            }
            if(tempcount2> count2){
                second_max_int= Math.abs(Intervals.elementAt(i));
                count2= tempcount2;
            }
            tempcount2= 0;
        }

        int max_int_occur= 0;
        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== max_int){
```

```
                max_int_occur++;
            }
        }

        int second_max_int_occur= 0;
        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== second_max_int){
                second_max_int_occur++;
            }
        }

        double rsomci= (double) second_max_int_occur/ max_int_occur;
        return rsomci;
    }

}
```

Bijlage III.18: RelativeStrengthOfTopPitch

```
import java.util.*;

/**
 * aandeel 2de meest voorkomende pitch / aandeel meest voorkomende
 * pitch
 *
 * @author Max
 *
 */

public class RelativeStrengthOfTopPitch{

    public static double getRelativeStrengthOfTopPitch(
    Vector<Integer> PedalPointVector){

        int common_pitch= 0;
        int second_common_pitch= 0;
        int count3= 0;
        int tempcount3= 0;
        int count4= 0;
        int tempcount4= 0;

        // Berekenen meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))){
                    tempcount3++;
                }
            }
            if(tempcount3> count3){
                common_pitch= Math.abs(PedalPointVector.elementAt(i));
                count3= tempcount3;
            }
            tempcount3= 0;
        }

        // Berekenen 2de meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))&&
                Math.abs(PedalPointVector.elementAt(i))!= common_pitch){
                    tempcount4++;
                }
            }
            if(tempcount4> count4){
                second_common_pitch=
                Math.abs(PedalPointVector.elementAt(i));
                count4= tempcount4;
            }
            tempcount4= 0;
        }

        int pitch_occur= 0;
        for(int i= 0; i< PedalPointVector.size(); i++){
```

```
            if(PedalPointVector.elementAt(i)== common_pitch){
                pitch_occur++;
            }
        }

        int second_pitch_occur= 0;
        for(int i= 0; i< PedalPointVector.size(); i++){
            if(PedalPointVector.elementAt(i)== second_common_pitch){
                second_pitch_occur++;
            }
        }
        double rsotp= (double) second_pitch_occur/ pitch_occur;
        return rsotp;
    }

}
```

Bijlage III.19: RepeatedNotes

```
import java.util.*;

/**
 * Berekent het aantal intervallen met grootte 0
 *
 * @author Max
 *
 */

public class RepeatedNotes{

    public static double getRepeatedNotes(Vector<Integer> Intervals){

        int rep_note= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Intervals.elementAt(i)== 0){
                rep_note++;
            }
        }

        double repeated_notes= (double) rep_note/ Intervals.size();
        return repeated_notes;
    }

}
```

Bijlage III.20: SizeOfMelodicArcs

```
package pedalpoints;

import java.util.*;

/**
 * berekent het gemiddelde interval tussen pieken en dalen
 *
 * @author Max
 *
 */

public class SizeOfMelodicArcs{

    public static double getSizeOfMelodicArcs(
    Vector<Integer> Intervals){

        int total_intervals= 0;
        int arcs= 1;
        double sign= Math.signum(Intervals.elementAt(0));

        // Bereken aantal arcs
        if(Intervals.elementAt(0)== 0){
            for(int i= 0; i< Intervals.size(); i++){
                sign= Math.signum(Intervals.elementAt(i));
                if(sign!= 0){
                    break;
                }
            }
        }

        for(int i= 0; i< Intervals.size()- 1; i++){
            if(Math.signum(Intervals.elementAt(i+ 1))!= 0&&
            sign!= Math.signum(Intervals.elementAt(i+ 1))){
                arcs++;
                sign= Math.signum(Intervals.elementAt(i+ 1));
            }
        }

        // berekenen totale interval
        for(int i= 0; i< Intervals.size(); i++){
            total_intervals+= Math.abs(Intervals.elementAt(i));
        }

        // Size of Melodic Arcs berekenen
        double sma= (double) total_intervals/ arcs;
        return sma;
    }

}
```

Bijlage III.21: StepwiseMotion

```
import java.util.*;

/**
 * Berekent het relatieve aantal stappen van 1 of 2
 *
 * @author Max
 *
 */

public class StepwiseMotion{

    public static double getStepwiseMotion(Vector<Integer> Intervals){

        int steps= 0;

        for(int i= 0; i< Intervals.size(); i++){
            if(Math.abs(Intervals.elementAt(i))== 1||
            Math.abs(Intervals.elementAt(i))== 2){
                steps++;
            }
        }

        double steps2= (double) steps/ Intervals.size();
        return steps2;

    }

}
```

Bijlage IV: Het project pedalpoints

Bijlage IV.1: Algo1CEM

```
package pedalpoints;

import java.util.*;

/**
 * First Descent Local Search Algoritme voor het optimaliseren van
 * PedalPointVector. Noten worden willekeurig en per 4 veranderd.
 * Indien de doelfunctiewaarde van de nieuwe oplossing beter is dan de
 * huidige doelfunctiewaarde, dan wordt deze direct aanvaardt als de
 * nieuwe 'beste' oplossing. Het proces doorloopt heel de notenvector
 * en wordt vervolgens ettelijke malen herhaald.
 *
 * Hier wordt tevens een Cross Entropy term in de doelfunctie
 * toegevoegd.
 *
 * @author Max
 *
 *         27 juni 2014
 *
 */

public class Algo1CEM{

    @SuppressWarnings("unchecked")
    public static Vector<Integer> berekenAlgo1CEM(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999;
        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        for(int i= 0; i< aantalrondes; i++){

            System.out.println("Beste Doelfunctiewaarde: "+
            bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1));

            for(int j= 0; j< aantalnoten; j+= 4){

                int r1= ENaturalMinor.length;
                Random random1= new Random();
                int index1= random1.nextInt(r1);
                int index2= random1.nextInt(r1);
                int index3= random1.nextInt(r1);
                int index4= random1.nextInt(r1);

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();
                PedalPointVector2.set(j, ENaturalMinor[index1]);
                PedalPointVector2.set(j+ 1, ENaturalMinor[index2]);
                PedalPointVector2.set(j+ 2, ENaturalMinor[index3]);
                PedalPointVector2.set(j+ 3, ENaturalMinor[index4]);
```

```
                IntervallenVector.clear();
                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                // Beperking die ervoor zorgt dat enkel het orgelpunt
                // na elkaar gespeeld kan worden, maar niet meer dan
                // 2 keer
                for(int k= 0; k< IntervallenVector.size(); k++){

                    if(IntervallenVector.elementAt(k)== 0&&
                    PedalPointVector2.elementAt(k)!= ZZ_MostCommonPitch
                    .getMostCommonPitch(PedalPointVector2)||
                    IntervallenVector.elementAt(k)== 0&&
                    IntervallenVector.elementAt(k)== IntervallenVector
                    .elementAt(Math.abs(k- 2))){

                        int index5= random1.nextInt(r1);
                        while(ENaturalMinor[index5]== PedalPointVector2
                        .elementAt(k)){
                            index5= random1.nextInt(r1);
                        }
                        PedalPointVector2.set(k,
                        ENaturalMinor[index5]);
                    }
                }

                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                doelfunctiewaarde=
                DoelFunctieCEM.berekenDoelFunctieCEM(
                PedalPointVector2, IntervallenVector, MarkovTrans);

                if(doelfunctiewaarde< bestedoelfunctiewaarde){

                    PedalPointVector=
                    (Vector<Integer>) PedalPointVector2.clone();
                    bestedoelfunctiewaarde= doelfunctiewaarde;
                }

            }
        }

        return PedalPointVector;
    }

}
```

Bijlage IV.2: Algo1

```
package pedalpoints;

import java.util.*;

/**
 * First Descent Local Search Algoritme voor het optimaliseren van
 * PedalPointVector. Noten worden willekeurig en per 4 veranderd.
 * Indien de doelfunctiewaarde van de nieuwe oplossing beter is dan de
 * huidige doelfunctiewaarde, dan wordt deze direct aanvaard als de
 * nieuwe 'beste' oplossing. Het proces doorloopt heel de notenvector
 * en wordt vervolgens ettelijke malen herhaald.
 *
 * @author Max 9 april 2014
 *
 */

public class Algo1LocalSearchFirstDescent0904{

    @SuppressWarnings("unchecked")
    public static Vector<Integer>
    berekenAlgo1LocalSearchFirstDescent0904(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999;
        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        for(int i= 0; i< aantalrondes; i++){

            System.out.println("Beste Doelfunctiewaarde: "+
            bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1));

            for(int j= 0; j< aantalnoten; j+= 4){

                int r1= ENaturalMinor.length;
                Random random1= new Random();
                int index1= random1.nextInt(r1);
                int index2= random1.nextInt(r1);
                int index3= random1.nextInt(r1);
                int index4= random1.nextInt(r1);

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();
                PedalPointVector2.set(j, ENaturalMinor[index1]);
                PedalPointVector2.set(j+ 1, ENaturalMinor[index2]);
                PedalPointVector2.set(j+ 2, ENaturalMinor[index3]);
                PedalPointVector2.set(j+ 3, ENaturalMinor[index4]);

                IntervallenVector.clear();
                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                // Beperking die ervoor zorgt dat enkel het orgelpunt
                // na elkaar gespeeld kan worden, maar niet meer dan
                // 2 keer
```

```
                for(int k= 0; k< IntervallenVector.size(); k++){

                    if(IntervallenVector.elementAt(k)== 0&&
                    PedalPointVector2.elementAt(k)!= ZZ_MostCommonPitch
                    .getMostCommonPitch(PedalPointVector2)||
                    IntervallenVector.elementAt(k)== 0&&
                    IntervallenVector.elementAt(k)== IntervallenVector
                    .elementAt(Math.abs(k- 2))){

                        int index5= random1.nextInt(r1);
                        while(ENaturalMinor[index5]== PedalPointVector2
                        .elementAt(k)){
                            index5= random1.nextInt(r1);
                        }
                        PedalPointVector2.set(k,
                        ENaturalMinor[index5]);
                    }
                }

                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                doelfunctiewaarde=
                DoelFunctie.berekenDoelFunctie(PedalPointVector2,
                IntervallenVector);

                if(doelfunctiewaarde< bestedoelfunctiewaarde){

                    PedalPointVector=
                    (Vector<Integer>) PedalPointVector2.clone();
                    bestedoelfunctiewaarde= doelfunctiewaarde;
                }

            }
        }

        return PedalPointVector;
    }

}
```

Bijlage IV.3: Algo1v2Iterator2Steps

```
package pedalpoints;

import java.util.*;

/**
 *
 * @author Max
 *
 */

public class Algo1v2Iterator2Steps{

    @SuppressWarnings("unchecked")
    public static Vector<Double> berekenAlgo1v2Iterator2steps(int j,
    int[] ENaturalMinor, Vector<Integer> PedalPointVector){

        int r1= ENaturalMinor.length;
        Random random1= new Random();
        int index1= random1.nextInt(r1);
        int index2= random1.nextInt(r1);

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        PedalPointVector2= (Vector<Integer>) PedalPointVector.clone();
        PedalPointVector2.set(j, ENaturalMinor[index1]);
        PedalPointVector2.set(j+ 1, ENaturalMinor[index2]);

        Vector<Integer> IntervallenVector= new Vector<Integer>(0);
        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        // Beperking die ervoor zorgt dat enkel het orgelpunt na
        // elkaar gespeeld kan worden, maar niet meer dan
        // 2 keer
        for(int k= 0; k< IntervallenVector.size(); k++){

            if(IntervallenVector.elementAt(k)== 0&&
            PedalPointVector2.elementAt(k)!= ZZ_MostCommonPitch
            .getMostCommonPitch(PedalPointVector2)||
            IntervallenVector.elementAt(k)== 0&&
            IntervallenVector.elementAt(k)== IntervallenVector
            .elementAt(Math.abs(k- 2))){

                int index5= random1.nextInt(r1);
                while(ENaturalMinor[index5]== PedalPointVector2
                .elementAt(k)){
                    index5= random1.nextInt(r1);
                }
                PedalPointVector2.set(k, ENaturalMinor[index5]);
            }
        }

        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        double doelfunctiewaarde=
        DoelFunctie.berekenDoelFunctie(PedalPointVector2,
        IntervallenVector);

        Vector<Double> PedalPointVector3= new Vector<Double>(0);
```

```
        PedalPointVector3= (Vector<Double>) PedalPointVector2.clone();
        Vector<Double> ValueHolder2Steps= new Vector<Double>(0);
        ValueHolder2Steps.add(0, doelfunctiewaarde);
        ValueHolder2Steps.addAll(1, PedalPointVector3);
        return ValueHolder2Steps;
    }

}
```

Bijlage IV.4: Algo1v2Iterator3Steps

```
package pedalpoints;

import java.util.*;

/**
 *
 * @author Max
 *
 */

public class Algo1v2Iterator3Steps{

    @SuppressWarnings("unchecked")
    public static Vector<Double> berekenAlgo1v2Iterator3steps(int j,
    int[] ENaturalMinor, Vector<Integer> PedalPointVector){

        int r1= ENaturalMinor.length;
        Random random1= new Random();
        int index1= random1.nextInt(r1);
        int index2= random1.nextInt(r1);
        int index3= random1.nextInt(r1);

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        PedalPointVector2= (Vector<Integer>) PedalPointVector.clone();
        PedalPointVector2.set(j, ENaturalMinor[index1]);
        PedalPointVector2.set(j+ 1, ENaturalMinor[index2]);
        PedalPointVector2.set(j+ 2, ENaturalMinor[index3]);

        Vector<Integer> IntervallenVector= new Vector<Integer>(0);
        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        // Beperking die ervoor zorgt dat enkel het orgelpunt na
        // elkaar gespeeld kan worden, maar niet meer dan
        // 2 keer
        for(int k= 0; k< IntervallenVector.size(); k++){

            if(IntervallenVector.elementAt(k)== 0&&
            PedalPointVector2.elementAt(k)!= ZZ_MostCommonPitch
            .getMostCommonPitch(PedalPointVector2)||
            IntervallenVector.elementAt(k)== 0&&
            IntervallenVector.elementAt(k)== IntervallenVector
            .elementAt(Math.abs(k- 2))){

                int index5= random1.nextInt(r1);
                while(ENaturalMinor[index5]== PedalPointVector2
                .elementAt(k)){
                    index5= random1.nextInt(r1);
                }
                PedalPointVector2.set(k, ENaturalMinor[index5]);
            }
        }

        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        double doelfunctiewaarde=
        DoelFunctie.berekenDoelFunctie(PedalPointVector2,
        IntervallenVector);
```

```
        Vector<Double> PedalPointVector3= new Vector<Double>(0);
        PedalPointVector3= (Vector<Double>) PedalPointVector2.clone();
        Vector<Double> ValueHolder3Steps= new Vector<Double>(0);
        ValueHolder3Steps.add(0, doelfunctiewaarde);
        ValueHolder3Steps.addAll(1, PedalPointVector3);
        return ValueHolder3Steps;
    }

}
```

Bijlage IV.5: Algo1v2Iterator4Steps

```
package pedalpoints;

import java.util.*;

/**
 *
 * @author Max
 *
 */

public class Algo1v2Iterator4Steps{

    @SuppressWarnings("unchecked")
    public static Vector<Double> berekenAlgo1v2Iterator4steps(int j,
    int[] ENaturalMinor, Vector<Integer> PedalPointVector){

        int r1= ENaturalMinor.length;
        Random random1= new Random();
        int index1= random1.nextInt(r1);
        int index2= random1.nextInt(r1);
        int index3= random1.nextInt(r1);
        int index4= random1.nextInt(r1);

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        PedalPointVector2= (Vector<Integer>) PedalPointVector.clone();
        PedalPointVector2.set(j, ENaturalMinor[index1]);
        PedalPointVector2.set(j+ 1, ENaturalMinor[index2]);
        PedalPointVector2.set(j+ 2, ENaturalMinor[index3]);
        PedalPointVector2.set(j+ 3, ENaturalMinor[index4]);

        Vector<Integer> IntervallenVector= new Vector<Integer>(0);
        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        // Beperking die ervoor zorgt dat enkel het orgelpunt na
        // elkaar gespeeld kan worden, maar niet meer dan
        // 2 keer
        for(int k= 0; k< IntervallenVector.size(); k++){

            if(IntervallenVector.elementAt(k)== 0&&
            PedalPointVector2.elementAt(k)!= ZZ_MostCommonPitch
            .getMostCommonPitch(PedalPointVector2)||
            IntervallenVector.elementAt(k)== 0&&
            IntervallenVector.elementAt(k)== IntervallenVector
            .elementAt(Math.abs(k- 2))){

                int index5= random1.nextInt(r1);
                while(ENaturalMinor[index5]== PedalPointVector2
                .elementAt(k)){
                    index5= random1.nextInt(r1);
                }
                PedalPointVector2.set(k, ENaturalMinor[index5]);
            }
        }

        IntervallenVector= Intervals.getIntervals(PedalPointVector2);

        double doelfunctiewaarde=
```

```
        DoelFunctie.berekenDoelFunctie(PedalPointVector2,
        IntervallenVector);

        Vector<Double> PedalPointVector3= new Vector<Double>(0);
        PedalPointVector3= (Vector<Double>) PedalPointVector2.clone();
        Vector<Double> ValueHolder4Steps= new Vector<Double>(0);
        ValueHolder4Steps.add(0, doelfunctiewaarde);
        ValueHolder4Steps.addAll(1, PedalPointVector3);
        return ValueHolder4Steps;
    }

}
```

Bijlage IV.6: Algo1v2

```
package pedalpoints;

import java.util.*;

/**
 * Algo 1v2
 *
 * Local Search Algoritme dat verschillende oplossingen afweegt. Per
 * iteratie j worden er resp. 2, 3 en 4 noten verandert in de
 * oplossing. De beste nieuwe oplossing wordt vergeleken met de
 * 'beste' globale oplossing. Is de nieuwe beter, dan wordt deze
 * direct aanvaard als de 'beste' oplossing. Dit wordt voor
 * verschillende indexen in de notenvector gedaan en vervolgens vele
 * malen herhaald.
 *
 * Gebaseerd op Algo1
 *
 * @author Max
 *
 *         10 april 2014
 *
 */

public class Algo1v2LocalSearchSteepestDescent1004{

    @SuppressWarnings("unchecked")
    public static Vector<Integer>
    berekenAlgo1v2LocalSearchSteepestDescent1004(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector){

        double bestedoelfunctiewaarde= 999999;

        Vector<Double> OplossingenVector= new Vector<Double>(0);

        Vector<Double> Values2Steps= new Vector<Double>(0);
        Vector<Double> Values3Steps= new Vector<Double>(0);
        Vector<Double> Values4Steps= new Vector<Double>(0);

        double doelfunctiewaarde1= 0;
        double doelfunctiewaarde2= 0;
        double doelfunctiewaarde3= 0;

        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int j1= 0;
        int j2= 0;
        int j3= 0;
        int j4= 0;
        int nochange= 0;
        int nochange2= 0;
        int nochangecounter= 0;

        for(int i= 0; i< aantalrondes; i++){

            System.out.println("Beste Doelfunctiewaarde: "+
```

```
bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+ " j1: "+ j1+
" j2: "+ j2+ " j3: "+ j3+ " j4: "+ j4);

j1= 0;
j2= 0;
j3= 0;
j4= 0;

int j= 0;
do{
    if(j+ 1< PedalPointVector.size()){

        Values2Steps=
        Algo1v2Iterator2Steps
        .berekenAlgo1v2Iterator2steps(j, ENaturalMinor,
        PedalPointVector);
        doelfunctiewaarde1= Values2Steps.elementAt(0);
        Values2Steps.removeElementAt(0);

    }else{
        doelfunctiewaarde1= 99999;
    }
    OplossingenVector.add(doelfunctiewaarde1);

    if(j+ 2< PedalPointVector.size()){

        Values3Steps=
        Algo1v2Iterator3Steps
        .berekenAlgo1v2Iterator3steps(j, ENaturalMinor,
        PedalPointVector);
        doelfunctiewaarde2= Values3Steps.elementAt(0);
        Values3Steps.removeElementAt(0);

    }else{
        doelfunctiewaarde2= 99999;
    }
    OplossingenVector.add(doelfunctiewaarde2);

    if(j+ 3< PedalPointVector.size()){

        Values4Steps=
        Algo1v2Iterator4Steps
        .berekenAlgo1v2Iterator4steps(j, ENaturalMinor,
        PedalPointVector);
        doelfunctiewaarde3= Values4Steps.elementAt(0);
        Values4Steps.removeElementAt(0);

    }else{
        doelfunctiewaarde3= 99999;
    }
    OplossingenVector.add(doelfunctiewaarde3);

    double doelfunctiewaarde=
    SmallestElement.getSmallest(OplossingenVector);

    if(doelfunctiewaarde< bestedoelfunctiewaarde){

        bestedoelfunctiewaarde= doelfunctiewaarde;
    }
```

```
    if(bestedoelfunctiewaarde== OplossingenVector
    .elementAt(0)){
        j+= 2;
        PedalPointVector=
        (Vector<Integer>) Values2Steps.clone();
        j2++;
    }else if(bestedoelfunctiewaarde== OplossingenVector
    .elementAt(1)){
        j+= 3;
        PedalPointVector=
        (Vector<Integer>) Values3Steps.clone();
        j3++;
    }else if(bestedoelfunctiewaarde== OplossingenVector
    .elementAt(2)){
        j+= 4;
        PedalPointVector=
        (Vector<Integer>) Values4Steps.clone();
        j4++;
    }else{
        j+=
        Math
        .min(4, Math.round((float) 0.05* aantalnoten));
        j1++;
    }

    OplossingenVector.clear();

}while(j< aantalnoten);

// shaker die willekeurig elementen verandert bij inertie
if(j2== 0&& j3== 0&& j4== 0){
    nochange++;
}
if(j3== 0&& j4== 0){
    nochange2++;
}
if(j2!= 0|| j3!= 0|| j4!= 0){
    nochange= 0;
    nochange2= 0;
}

if(nochange2> 15+ 40/ aantalnoten||
nochange> (15+ 40/ aantalnoten)/ 2){

    OplossingenMagazijn1.add(bestedoelfunctiewaarde);
    OplossingenMagazijn2.add(PedalPointVector);

    for(int k= 0; k< (double) aantalnoten/ 10; k++){

        int r= ENaturalMinor.length;
        Random random= new Random();
        int index= random.nextInt(r);

        int rp= PedalPointVector.size();
        int indexp= random.nextInt(rp);

        PedalPointVector
        .set(indexp, ENaturalMinor[index]);
```

```
                }

                Vector<Integer> Intervallen= new Vector<Integer>(0);
                Intervallen= Intervals.getIntervals(PedalPointVector);
                bestedoelfunctiewaarde=
                DoelFunctie.berekenDoelFunctie(PedalPointVector,
                Intervallen);
                nochange= 0;
                nochangecounter++;
            }

            // Doet er een aantal rondes bij indien een kritieke
            // waarde niet overschreden wordt.
            if(i== (aantalrondes- 1)){
                if(bestedoelfunctiewaarde< 1.1* SmallestElement
                .getSmallest(OplossingenMagazijn1)){
                    aantalrondes+= 10;
                }
            }
        }// einde aantal rondes

        // kijken of de beste oplossing al dan niet de laatste beste
        // oplossing is
        if(SmallestElement.getSmallest(OplossingenMagazijn1)<
             bestedoelfunctiewaarde){

            bestedoelfunctiewaarde=
            SmallestElement.getSmallest(OplossingenMagazijn1);
            int minnie=
            OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
            PedalPointVector= OplossingenMagazijn2.elementAt(minnie);
        }

        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return PedalPointVector;
    }// einde method

}// einde class
```

Bijlage IV.7: Algo2

```
package pedalpoints;

import java.util.*;

/**
 * Hetzelfde als Algo1, maar met alle mogelijke combinaties ipv random
 * generator
 *
 * @author Max 13 april 2014
 *
 */

public class Algo2{

    @SuppressWarnings("unchecked")
    public static Vector<Integer> berekenAlgo2(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999;
        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        int r1= ENaturalMinor.length;
        Random random1= new Random();

        int incr= 2;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();

                for(int k= 0; k< ENaturalMinor.length; k++){
                    for(int l= 0; l< ENaturalMinor.length; l++){
                        for(int m= 0; m< ENaturalMinor.length; m++){
                            for(int n= 0; n< ENaturalMinor.length; n++){

                                if(j+ 3< aantalnoten){

                                    PedalPointVector2.set(j,
                                    ENaturalMinor[k]);
                                    PedalPointVector2.set(j+ 1,
                                    ENaturalMinor[l]);
                                    PedalPointVector2.set(j+ 2,
                                    ENaturalMinor[m]);
                                    PedalPointVector2.set(j+ 3,
                                    ENaturalMinor[n]);
                                }else if(j+ 2< aantalnoten){
                                    PedalPointVector2.set(j,
                                    ENaturalMinor[l]);
                                    PedalPointVector2.set(j+ 1,
                                    ENaturalMinor[m]);
                                    PedalPointVector2.set(j+ 2,
```

```
    ENaturalMinor[n]);
}else if(j+ 1< aantalnoten){
    PedalPointVector2.set(j,
    ENaturalMinor[m]);
    PedalPointVector2.set(j+ 1,
    ENaturalMinor[n]);
}else{
    PedalPointVector2.set(j,
    ENaturalMinor[n]);
}

IntervallenVector.clear();
IntervallenVector=
Intervals
.getIntervals(PedalPointVector2);

// Beperking die ervoor zorgt dat
// enkel het orgelpunt na elkaar
// gespeeld kan worden,
// maar niet meer dan 2 keer
for(int z= 0; z< IntervallenVector
.size(); z++){

    if(IntervallenVector.elementAt(z)== 0&&
    PedalPointVector2.elementAt(z)!=
       ZZ_MostCommonPitch
    .getMostCommonPitch(PedalPointVector2)||
    IntervallenVector.elementAt(z)== 0&&
    IntervallenVector.elementAt(z)==
       IntervallenVector
    .elementAt(Math.abs(z- 2))){

        int index5=
        random1.nextInt(r1);
        while(ENaturalMinor[index5]==
             PedalPointVector2
        .elementAt(z)){
            index5=
            random1.nextInt(r1);
        }
        PedalPointVector2.set(z,
        ENaturalMinor[index5]);
    }
}

IntervallenVector=
Intervals
.getIntervals(PedalPointVector2);

doelfunctiewaarde=
DoelFunctie.berekenDoelFunctie(
PedalPointVector2, IntervallenVector);

if(doelfunctiewaarde< bestedoelfunctiewaarde){
    bestedoelfunctiewaarde=
    doelfunctiewaarde;
    PedalPointVector=
    (Vector<Integer>) PedalPointVector2
    .clone();
```

```
                                }

                            }// n
                        }// m
                    }// l
                }// k

                System.out.println("Beste Doelfunctiewaarde: "+
                bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
                ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

            }// j
        }// i

        return PedalPointVector;
    }// method

}// class
```

Bijlage IV.8: Algo2v2

```
package pedalpoints;

import java.util.Random;
import java.util.Vector;

/**
 * Hetzelfde als Algo2, maar dan met kleinere stappen teneinde de
 * combinatiemogelijkheden drastisch te bepereken
 *
 * @author Max 13 april 2014
 *
 */
public class Algo2v2{

    @SuppressWarnings("unchecked")
    public static Vector<Integer> berekenAlgo2v2(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999;

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int r1= ENaturalMinor.length;
        int rp= PedalPointVector.size();
        Random random1= new Random();

        int nochangecounter= 0;

        int incr= 2;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();

                for(int k= 0; k< ENaturalMinor.length; k++){
                    for(int l= 0; l< ENaturalMinor.length; l++){

                        if(j+ 1< aantalnoten){

                            PedalPointVector2
                            .set(j, ENaturalMinor[k]);
                            PedalPointVector2.set(j+ 1,
                            ENaturalMinor[l]);
                        }else{
                            PedalPointVector2
                            .set(j, ENaturalMinor[l]);
```

```
                }

                IntervallenVector.clear();
                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                // Beperking die ervoor zorgt dat enkel het
                // orgelpunt na elkaar gespeeld kan worden,
                // maar niet meer dan 2 keer
                for(int z= 0; z< IntervallenVector.size(); z++){

                    if(IntervallenVector.elementAt(z)== 0&&
                    PedalPointVector2.elementAt(z)!= ZZ_MostCommonPitch
                    .getMostCommonPitch(PedalPointVector2)||
                    IntervallenVector.elementAt(z)== 0&&
                    IntervallenVector.elementAt(z)== IntervallenVector
                    .elementAt(Math.abs(z- 2))){

                        int index5= random1.nextInt(r1);
                        while(ENaturalMinor[index5]== PedalPointVector2
                        .elementAt(z)){
                            index5= random1.nextInt(r1);
                        }
                        PedalPointVector2.set(z,
                        ENaturalMinor[index5]);
                    }
                }

                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                doelfunctiewaarde=
                DoelFunctie.berekenDoelFunctie(
                PedalPointVector2, IntervallenVector);

                if(doelfunctiewaarde< bestedoelfunctiewaarde){
                    bestedoelfunctiewaarde= doelfunctiewaarde;
                    PedalPointVector=
                    (Vector<Integer>) PedalPointVector2
                    .clone();
                }

            }// l
        }// k

        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
        ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

    }// j

    WaardeNaRonde.add(bestedoelfunctiewaarde);

    // shaker die willekeurig elementen verandert bij inertie
    if(i> 0){
        double verandering=
        WaardeNaRonde.elementAt(i)-
        WaardeNaRonde.elementAt(i- 1);
        if(verandering== 0){
```

```
                    OplossingenMagazijn1.add(bestedoelfunctiewaarde);
                    OplossingenMagazijn2.add(PedalPointVector);

                    for(int n= 0; n< aantalnoten/ 10; n++){

                        int index= random1.nextInt(r1);
                        int indexp= random1.nextInt(rp);

                        PedalPointVector.set(indexp,
                        ENaturalMinor[index]);
                    }

                    IntervallenVector=
                    Intervals.getIntervals(PedalPointVector);
                    bestedoelfunctiewaarde=
                    DoelFunctie.berekenDoelFunctie(PedalPointVector,
                    IntervallenVector);

                    nochangecounter++;
                }
            }

            // Doet er een aantal rondes bij indien een kritieke
            // waarde niet overschreden wordt.
            if(i== (aantalrondes- 1)){
                if(bestedoelfunctiewaarde< 1.1* SmallestElement
                .getSmallest(OplossingenMagazijn1)){
                    aantalrondes+= 10;
                }
            }
        }// i

        // kijken of de beste oplossing al dan niet de laatste beste
        // oplossing is
        if(SmallestElement.getSmallest(OplossingenMagazijn1)<
             bestedoelfunctiewaarde){

            bestedoelfunctiewaarde=
            SmallestElement.getSmallest(OplossingenMagazijn1);
            int minnie=
            OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
            PedalPointVector= OplossingenMagazijn2.elementAt(minnie);
        }

        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return PedalPointVector;
    }// method
}
```

Bijlage IV.9: Algo2v2CEM

```
package pedalpoints;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Hetzelfde als Algo2, maar dan met kleinere stappen teneinde de
 * combinatiemogelijkheden drastisch te beperken. Deze maakt gebruik
 * van een Cross Entropy term in de doelfunctie.
 *
 * @author Max 26 juni 2014
 *
 */
public class Algo2v2CEM{

    @SuppressWarnings("unchecked")
    public static Vector<Integer> berekenAlgo2v2CEM(int aantalrondes,
    int aantalnoten, int[] ENaturalMinor,
    Vector<Integer> PedalPointVector,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 99999999;

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int r1= ENaturalMinor.length;
        int rp= PedalPointVector.size();
        Random random1= new Random();

        int nochangecounter= 0;

        int incr= 1;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();

                for(int k= 0; k< ENaturalMinor.length; k++){
                    for(int l= 0; l< ENaturalMinor.length; l++){

                        if(j+ 1< aantalnoten){

                            PedalPointVector2
                            .set(j, ENaturalMinor[k]);
                            PedalPointVector2.set(j+ 1,
                            ENaturalMinor[l]);
```

```
                }else{
                    PedalPointVector2
                    .set(j, ENaturalMinor[l]);
                }

                IntervallenVector.clear();
                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                // Beperking die ervoor zorgt dat enkel het
                // orgelpunt na elkaar gespeeld kan worden,
                // maar niet meer dan 2 keer
                for(int z= 0; z< IntervallenVector.size(); z++){

                    if(IntervallenVector.elementAt(z)== 0&&
                    PedalPointVector2.elementAt(z)!= ZZ_MostCommonPitch
                    .getMostCommonPitch(PedalPointVector2)||
                    IntervallenVector.elementAt(z)== 0&&
                    IntervallenVector.elementAt(z)== IntervallenVector
                    .elementAt(Math.abs(z- 2))){

                        int index5= random1.nextInt(r1);
                        while(ENaturalMinor[index5]== PedalPointVector2
                        .elementAt(z)){
                            index5= random1.nextInt(r1);
                        }
                        PedalPointVector2.set(z,
                        ENaturalMinor[index5]);
                    }
                }

                IntervallenVector=
                Intervals.getIntervals(PedalPointVector2);

                doelfunctiewaarde=
                DoelFunctieCEM.berekenDoelFunctieCEM(
                PedalPointVector2, IntervallenVector,
                MarkovTrans);

                if(doelfunctiewaarde< bestedoelfunctiewaarde){
                    bestedoelfunctiewaarde= doelfunctiewaarde;
                    PedalPointVector=
                    (Vector<Integer>) PedalPointVector2
                    .clone();
                }

            }// l
        }// k

        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
        ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

    }// j

    WaardeNaRonde.add(bestedoelfunctiewaarde);

    // shaker die willekeurig elementen verandert bij inertie
    if(i> 0){
```

```
                double verandering=
                WaardeNaRonde.elementAt(i)-
                WaardeNaRonde.elementAt(i- 1);
                if(verandering== 0){

                    OplossingenMagazijn1.add(bestedoelfunctiewaarde);
                    OplossingenMagazijn2.add(PedalPointVector);

                    for(int n= 0; n< aantalnoten/ 10; n++){

                        int index= random1.nextInt(r1);
                        int indexp= random1.nextInt(rp);

                        PedalPointVector.set(indexp,
                        ENaturalMinor[index]);
                    }

                    IntervallenVector=
                    Intervals.getIntervals(PedalPointVector);
                    bestedoelfunctiewaarde=
                    DoelFunctieCEM.berekenDoelFunctieCEM(
                    PedalPointVector, IntervallenVector, MarkovTrans);

                    nochangecounter++;
                }
            }

            // Doet er een aantal rondes bij indien een kritieke
            // waarde niet overschreden wordt.
            if(i== (aantalrondes- 1)){
                System.out.println("bdfw: "+ bestedoelfunctiewaarde);
                if(bestedoelfunctiewaarde< 1.1* SmallestElement
                .getSmallest(OplossingenMagazijn1)){
                    aantalrondes+= 10;
                }
            }
        }// i

        // kijken of de beste oplossing al dan niet de laatste beste
        // oplossing is
        if(SmallestElement.getSmallest(OplossingenMagazijn1)<
             bestedoelfunctiewaarde){

            bestedoelfunctiewaarde=
            SmallestElement.getSmallest(OplossingenMagazijn1);
            int minnie=
            OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
            PedalPointVector= OplossingenMagazijn2.elementAt(minnie);
        }

        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return PedalPointVector;
    }// method

}
```

Bijlage IV.10: Algo2v2CEM2

```
package pedalpoints;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Hetzelfde als Algo2, maar dan met kleinere stappen teneinde de
 * combinatiemogelijkheden drastisch te beperken. Deze maakt gebruik
 * van een Cross Entropy term in de doelfunctie.
 *
 * De Cross Entropy term wordt berekend op basis van een Markov
 * transitiematrix van de eerste orde en een van de 2de orde.
 *
 * @author Max
 *
 *         27 juni 2014
 *
 */
public class Algo2v2CEM2{

    @SuppressWarnings("unchecked")
    public static
    Vector<Integer>
    berekenAlgo2v2CEM2(
    int aantalrondes,
    int aantalnoten,
    int[] ENaturalMinor,
    Vector<Integer> PedalPointVector,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999999;

        Vector<Integer> PedalPointVector2=
        new Vector<Integer>(PedalPointVector.size());
        Vector<Integer> IntervallenVector= new Vector<Integer>(0);

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int r1= ENaturalMinor.length;
        int rp= PedalPointVector.size();
        Random random1= new Random();

        int nochangecounter= 0;

        int incr= 2;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){

                PedalPointVector2=
                (Vector<Integer>) PedalPointVector.clone();
```

```
for(int k= 0; k< ENaturalMinor.length; k++){
    for(int l= 0; l< ENaturalMinor.length; l++){

        if(j+ 1< aantalnoten){

            PedalPointVector2
            .set(j, ENaturalMinor[k]);
            PedalPointVector2.set(j+ 1,
            ENaturalMinor[l]);
        }else{
            PedalPointVector2
            .set(j, ENaturalMinor[l]);
        }

        IntervallenVector.clear();
        IntervallenVector=
        Intervals.getIntervals(PedalPointVector2);

        // Beperking die ervoor zorgt dat enkel het
        // orgelpunt na elkaar gespeeld kan worden,
        // maar niet meer dan 2 keer
        for(int z= 0; z< IntervallenVector.size(); z++){

            if(IntervallenVector.elementAt(z)== 0&&
            PedalPointVector2.elementAt(z)!= ZZ_MostCommonPitch
            .getMostCommonPitch(PedalPointVector2)||
            IntervallenVector.elementAt(z)== 0&&
            IntervallenVector.elementAt(z)== IntervallenVector
            .elementAt(Math.abs(z- 2))){

                int index5= random1.nextInt(r1);
                while(ENaturalMinor[index5]== PedalPointVector2
                .elementAt(z)){
                    index5= random1.nextInt(r1);
                }
                PedalPointVector2.set(z,
                ENaturalMinor[index5]);
            }
        }

        IntervallenVector=
        Intervals.getIntervals(PedalPointVector2);

        doelfunctiewaarde=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(
        PedalPointVector2, IntervallenVector,
        MarkovTrans, MarkovTrans2);

        if(doelfunctiewaarde< bestedoelfunctiewaarde){
            bestedoelfunctiewaarde= doelfunctiewaarde;
            PedalPointVector=
            (Vector<Integer>) PedalPointVector2
            .clone();
        }

    }// l
}// k
```

```
            System.out.println("Beste Doelfunctiewaarde: "+
            bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
            ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

        }// j

        WaardeNaRonde.add(bestedoelfunctiewaarde);

        // shaker die willekeurig elementen verandert bij inertie
        if(i> 0){
            double verandering=
            WaardeNaRonde.elementAt(i)-
            WaardeNaRonde.elementAt(i- 1);
            if(verandering== 0){

                OplossingenMagazijn1.add(bestedoelfunctiewaarde);
                OplossingenMagazijn2.add(PedalPointVector);

                for(int n= 0; n< aantalnoten/ 10; n++){

                    int index= random1.nextInt(r1);
                    int indexp= random1.nextInt(rp);

                    PedalPointVector.set(indexp,
                    ENaturalMinor[index]);
                }

                IntervallenVector=
                Intervals.getIntervals(PedalPointVector);
                bestedoelfunctiewaarde=
                DoelFunctieCEM2.berekenDoelFunctieCEM2(
                PedalPointVector, IntervallenVector, MarkovTrans,
                MarkovTrans2);

                nochangecounter++;
            }
        }

        // Doet er een aantal rondes bij indien een kritieke
        // waarde niet overschreden wordt.
        if(i== (aantalrondes- 1)){
            if(bestedoelfunctiewaarde< 1.1* SmallestElement
            .getSmallest(OplossingenMagazijn1)){
                aantalrondes+= 10;
            }
        }
    }// i

    // kijken of de beste oplossing al dan niet de laatste beste
    // oplossing is
    if(SmallestElement.getSmallest(OplossingenMagazijn1)<
         bestedoelfunctiewaarde){

        bestedoelfunctiewaarde=
        SmallestElement.getSmallest(OplossingenMagazijn1);
        int minnie=
        OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
        PedalPointVector= OplossingenMagazijn2.elementAt(minnie);
    }
```

```
        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return PedalPointVector;
    }// method

}
```

Bijlage IV.11: DoelFunctie

```
package pedalpoints;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * @author Max
 *
 */

public class DoelFunctie{

    public static double berekenDoelFunctie(
    Vector<Integer> PedalPointVector, Vector<Integer> Intervals){

        // Vector maken met de berekende referentiewaarden
        Vector<Double> ReferentieVector= new Vector<Double>(0);
        double[] RefVar=
        new double[]{0.5423, 5.831, 0.05527, 0.5217, 3.5, 1.0801,
        7.125, 0.1125, 0.07863, 0.1565, 0.005988, 4.625, 0.2529,
        0.398, 4.875, 3.375, 0.8032, 0.4462, 0.1002/**/, 6.87,
        0.1771};

        Vector<Double> RefVar2= new Vector<Double>(0);
        for(int i= 0; i< RefVar.length; i++){
            RefVar2.add(RefVar[i]);
        }
        ReferentieVector.addAll(RefVar2);

        // Vector maken voor waarden van berekende variabelen
        // (jSymbolic gebaseerd)
        Vector<Double> jSymVar= new Vector<Double>(0);

        double arpeg=
        AmountOfArpeggiation.getAmountOfArpeggiation(Intervals);
        jSymVar.add(arpeg);

        double avmi=
        AverageMelodicInterval.getAverageMelodicInterval(Intervals);
        jSymVar.add(avmi);

        double chromes= ChromaticMotion.getChromaticMotion(Intervals);
        jSymVar.add(chromes);

        double dm= DirectionOfMotion.getDirectionOfMotion(Intervals);
        jSymVar.add(dm);

        double discommonint=
        DistanceBetweenMostCommonMelodicIntervals
        .getDistanceBetweenMostCommonMelodicIntervals(Intervals);
        jSymVar.add(discommonint);

        double doma=
        DurationOfMelodicArcs.getDurationOfMelodicArcs(
        PedalPointVector, Intervals);
```

```
jSymVar.add(doma);

double inbesp=
IntervalBetweenStrongestPitches
.getIntervalBetweenStrongestPitches(PedalPointVector);
jSymVar.add(inbesp);

double fifthz= MelodicFifths.getMelodicFifths(Intervals);
jSymVar.add(fifthz);

double octavez= MelodicOctaves.getMelodicOctaves(Intervals);
jSymVar.add(octavez);

double thirdz= MelodicThirds.getMelodicThirds(Intervals);
jSymVar.add(thirdz);

double tritonez=
MelodicTritones.getMelodicTritones(Intervals);
jSymVar.add(tritonez);

double commelint=
MostCommonMelodicInterval
.getMostCommonMelodicInterval(Intervals);
jSymVar.add(commelint);

double commelintprev=
MostCommonMelodicIntervalPrevalence
.getMostCommonMelodicIntervalPrevalence(Intervals);
jSymVar.add(commelintprev);

double compitprev=
MostCommonPitchPrevalence
.getMostCommonPitchPrevalence(PedalPointVector);
jSymVar.add(compitprev);

double ncommint=
NumberOfCommonMelodicIntervals
.getNumberOfCommonMelodicIntervals(Intervals);
jSymVar.add(ncommint);

double numcompit=
NumberOfCommonPitches
.getNumberOfCommonPitches(PedalPointVector);
jSymVar.add(numcompit);

double relstrcommint=
RelativeStrengthOfMostCommonIntervals
.getRelativeStrengthOfMostCommonIntervals(Intervals);
jSymVar.add(relstrcommint);

double relstrtopitch=
RelativeStrengthOfTopPitch
.getRelativeStrengthOfTopPitch(PedalPointVector);
jSymVar.add(relstrtopitch);

double repnote=
100* RepeatedNotes.getRepeatedNotes(Intervals);// Pas Op
jSymVar.add(repnote);
```

```
        double sizarcs=
        SizeOfMelodicArcs.getSizeOfMelodicArcs(Intervals);
        jSymVar.add(sizarcs);

        double stepmo= StepwiseMotion.getStepwiseMotion(Intervals);
        jSymVar.add(stepmo);

        // berekenen van de doelfunctiewaarde
        double doelfunctiewaarde= 0;
        for(int i= 0; i< jSymVar.size(); i++){

            doelfunctiewaarde+=
            (double) Math
            .abs((double) (jSymVar.elementAt(i)- ReferentieVector
            .elementAt(i))/ ReferentieVector.elementAt(i));
        }

        return doelfunctiewaarde;

    }// einde method

}// einde class
```

Bijlage IV.12: DoelFunctieCEM

```
package pedalpoints;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * Bijkomend is er in de doelfunctie een soort van Cross Entropy term
 * gemaakt die een fout toevoegd gebaseerd op een Markov
 * transitiematrix.
 *
 * @author Max
 *
 */

public class DoelFunctieCEM{

    public static double berekenDoelFunctieCEM(
    Vector<Integer> PedalPointVector, Vector<Integer> Intervals,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans){

        // Vector maken met de berekende referentiewaarden
        Vector<Double> ReferentieVector= new Vector<Double>(0);
        double[] RefVar=
        new double[]{0.5423, 5.831, 0.05527, 0.5217, 3.5, 1.0801,
        7.125, 0.1125, 0.07863, 0.1565, 0.005988, 4.625, 0.2529,
        0.398, 4.875, 3.375, 0.8032, 0.4462, 0.1002/**/, 6.87,
        0.1771};

        Vector<Double> RefVar2= new Vector<Double>(0);
        for(int i= 0; i< RefVar.length; i++){
            RefVar2.add(RefVar[i]);
        }
        ReferentieVector.addAll(RefVar2);

        // Vector maken voor waarden van berekende variabelen
        // (jSymbolic gebaseerd)
        Vector<Double> jSymVar= new Vector<Double>(0);

        double arpeg=
        AmountOfArpeggiation.getAmountOfArpeggiation(Intervals);
        jSymVar.add(arpeg);

        double avmi=
        AverageMelodicInterval.getAverageMelodicInterval(Intervals);
        jSymVar.add(avmi);

        double chromes= ChromaticMotion.getChromaticMotion(Intervals);
        jSymVar.add(chromes);

        double dm= DirectionOfMotion.getDirectionOfMotion(Intervals);
        jSymVar.add(dm);

        double discommonint=
        DistanceBetweenMostCommonMelodicIntervals
        .getDistanceBetweenMostCommonMelodicIntervals(Intervals);
```

```
jSymVar.add(discommonint);

double doma=
DurationOfMelodicArcs.getDurationOfMelodicArcs(
PedalPointVector, Intervals);
jSymVar.add(doma);

double inbesp=
IntervalBetweenStrongestPitches
.getIntervalBetweenStrongestPitches(PedalPointVector);
jSymVar.add(inbesp);

double fifthz= MelodicFifths.getMelodicFifths(Intervals);
jSymVar.add(fifthz);

double octavez= MelodicOctaves.getMelodicOctaves(Intervals);
jSymVar.add(octavez);

double thirdz= MelodicThirds.getMelodicThirds(Intervals);
jSymVar.add(thirdz);

double tritonez=
MelodicTritones.getMelodicTritones(Intervals);
jSymVar.add(tritonez);

double commelint=
MostCommonMelodicInterval
.getMostCommonMelodicInterval(Intervals);
jSymVar.add(commelint);

double commelintprev=
MostCommonMelodicIntervalPrevalence
.getMostCommonMelodicIntervalPrevalence(Intervals);
jSymVar.add(commelintprev);

double compitprev=
MostCommonPitchPrevalence
.getMostCommonPitchPrevalence(PedalPointVector);
jSymVar.add(compitprev);

double ncommint=
NumberOfCommonMelodicIntervals
.getNumberOfCommonMelodicIntervals(Intervals);
jSymVar.add(ncommint);

double numcompit=
NumberOfCommonPitches
.getNumberOfCommonPitches(PedalPointVector);
jSymVar.add(numcompit);

double relstrcommint=
RelativeStrengthOfMostCommonIntervals
.getRelativeStrengthOfMostCommonIntervals(Intervals);
jSymVar.add(relstrcommint);

double relstrtopitch=
RelativeStrengthOfTopPitch
.getRelativeStrengthOfTopPitch(PedalPointVector);
jSymVar.add(relstrtopitch);
```

```
        double repnote=
        100* RepeatedNotes.getRepeatedNotes(Intervals);// Pas Op
        jSymVar.add(repnote);

        double sizarcs=
        SizeOfMelodicArcs.getSizeOfMelodicArcs(Intervals);
        jSymVar.add(sizarcs);

        double stepmo= StepwiseMotion.getStepwiseMotion(Intervals);
        jSymVar.add(stepmo);

        // Berekenen van de procentuele afwijkingen op de jSymbolic
        // gebaseerde referentiewaarden.
        double doelfunctiewaarde= 0;
        for(int i= 0; i< jSymVar.size(); i++){

            doelfunctiewaarde+=
            (double) Math
            .abs((double) (jSymVar.elementAt(i)- ReferentieVector
            .elementAt(i))/ ReferentieVector.elementAt(i));
        }

        // Cross Entropy fout toevoegen aan doelfunctie.
        double TransProb= 0;
        double CEfout= 0;

        for(int i= 0; i< Intervals.size()- 1; i++){

            try{
                TransProb=
                MarkovTrans.get(Intervals.elementAt(i)).get(
                Intervals.elementAt(i+ 1));
            }catch(NullPointerException e){
                TransProb= 0;
            }// Een catch voor het geval een intervalpaar niet in de
             // transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb< 0.000001){
                CEfout= 6;
            }else{
                CEfout= Math.log10(1/ TransProb);
            }

            doelfunctiewaarde+= CEfout;
        }

        return doelfunctiewaarde;

    }// einde method

}// einde class
```

Bijlage IV.13: DoelFunctieCEM2

```
package pedalpoints;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * Bijkomend is er in de doelfunctie een soort van Cross Entropy term
 * gemaakt die een fout toevoegd gebaseerd op een Markov
 * transitiematrix.
 *
 * In deze doelfunctie is een Markov transitiematrix van zowel de
 * eerste als de tweede orde verwerkt.
 *
 * @author Max
 *
 */

public class DoelFunctieCEM2{

    public static
    double
    berekenDoelFunctieCEM2(
    Vector<Integer> PedalPointVector,
    Vector<Integer> Intervals,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2){

        // Vector maken met de berekende referentiewaarden
        Vector<Double> ReferentieVector= new Vector<Double>(0);
        double[] RefVar=
        new double[]{0.5423, 5.831, 0.05527, 0.5217, 3.5, 1.0801,
        7.125, 0.1125, 0.07863, 0.1565, 0.005988, 4.625, 0.2529,
        0.398, 4.875, 3.375, 0.8032, 0.4462, 0.1002/**/, 6.87,
        0.1771};

        Vector<Double> RefVar2= new Vector<Double>(0);
        for(int i= 0; i< RefVar.length; i++){
            RefVar2.add(RefVar[i]);
        }
        ReferentieVector.addAll(RefVar2);

        // Vector maken voor waarden van berekende variabelen
        // (jSymbolic gebaseerd)
        Vector<Double> jSymVar= new Vector<Double>(0);

        double arpeg=
        AmountOfArpeggiation.getAmountOfArpeggiation(Intervals);
        jSymVar.add(arpeg);

        double avmi=
        AverageMelodicInterval.getAverageMelodicInterval(Intervals);
        jSymVar.add(avmi);

        double chromes= ChromaticMotion.getChromaticMotion(Intervals);
        jSymVar.add(chromes);
```

```
double dm= DirectionOfMotion.getDirectionOfMotion(Intervals);
jSymVar.add(dm);

double discommonint=
DistanceBetweenMostCommonMelodicIntervals
.getDistanceBetweenMostCommonMelodicIntervals(Intervals);
jSymVar.add(discommonint);

double doma=
DurationOfMelodicArcs.getDurationOfMelodicArcs(
PedalPointVector, Intervals);
jSymVar.add(doma);

double inbesp=
IntervalBetweenStrongestPitches
.getIntervalBetweenStrongestPitches(PedalPointVector);
jSymVar.add(inbesp);

double fifthz= MelodicFifths.getMelodicFifths(Intervals);
jSymVar.add(fifthz);

double octavez= MelodicOctaves.getMelodicOctaves(Intervals);
jSymVar.add(octavez);

double thirdz= MelodicThirds.getMelodicThirds(Intervals);
jSymVar.add(thirdz);

double tritonez=
MelodicTritones.getMelodicTritones(Intervals);
jSymVar.add(tritonez);

double commelint=
MostCommonMelodicInterval
.getMostCommonMelodicInterval(Intervals);
jSymVar.add(commelint);

double commelintprev=
MostCommonMelodicIntervalPrevalence
.getMostCommonMelodicIntervalPrevalence(Intervals);
jSymVar.add(commelintprev);

double compitprev=
MostCommonPitchPrevalence
.getMostCommonPitchPrevalence(PedalPointVector);
jSymVar.add(compitprev);

double ncommint=
NumberOfCommonMelodicIntervals
.getNumberOfCommonMelodicIntervals(Intervals);
jSymVar.add(ncommint);

double numcompit=
NumberOfCommonPitches
.getNumberOfCommonPitches(PedalPointVector);
jSymVar.add(numcompit);

double relstrcommint=
RelativeStrengthOfMostCommonIntervals
```

```
.getRelativeStrengthOfMostCommonIntervals(Intervals);
jSymVar.add(relstrcommint);

double relstrtopitch=
RelativeStrengthOfTopPitch
.getRelativeStrengthOfTopPitch(PedalPointVector);
jSymVar.add(relstrtopitch);

double repnote=
100* RepeatedNotes.getRepeatedNotes(Intervals);// Pas Op
jSymVar.add(repnote);

double sizarcs=
SizeOfMelodicArcs.getSizeOfMelodicArcs(Intervals);
jSymVar.add(sizarcs);

double stepmo= StepwiseMotion.getStepwiseMotion(Intervals);
jSymVar.add(stepmo);

// Berekenen van de procentuele afwijkingen op de jSymbolic
// gebaseerde referentiewaarden.
double doelfunctiewaarde= 0;
for(int i= 0; i< jSymVar.size(); i++){

    doelfunctiewaarde+=
    (double) Math
    .abs((double) (jSymVar.elementAt(i)- ReferentieVector
    .elementAt(i))/ ReferentieVector.elementAt(i));
}

// Cross Entropy fout toevoegen aan doelfunctie.

// Markov 1ste orde
double TransProb= 0;
double CEfout= 0;

for(int i= 0; i< Intervals.size()- 1; i++){

    try{
        TransProb=
        MarkovTrans.get(Intervals.elementAt(i)).get(
        Intervals.elementAt(i+ 1));
    }catch(NullPointerException e){
        TransProb= 0;
    }// Een catch voor het geval een intervalpaar niet in de
     // transitiematrix aanwezig is.

    // lim x-->0 log(1/x) = oneindig vermijden!
    if(TransProb< 0.000001){
        CEfout= 6;
    }else{
        CEfout= Math.log10(1/ TransProb);
    }

    doelfunctiewaarde+= 0.2* CEfout;
}

// Markov 2de orde
double TransProb2= 0;
```

```
        double CEfout2= 0;

        for(int i= 0; i< Intervals.size()- 2; i++){

            try{
                TransProb2=
                MarkovTrans2.get(Intervals.elementAt(i))
                .get(Intervals.elementAt(i+ 1))
                .get(Intervals.elementAt(i+ 2));
            }catch(NullPointerException e){
                TransProb2= 0;
            }// Een catch voor het geval een intervaltriplet niet in
             // de transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb2< 0.000001){
                CEfout2= 6;
            }else{
                CEfout2= Math.log10(1/ TransProb2);
            }

            doelfunctiewaarde+= 0.2* CEfout2;
        }

        return doelfunctiewaarde;

    }// einde method

}// einde class
```

Bijlage IV.14: Intervals

```
package pedalpoints;

import java.util.*;

/**
 * berekent de vector met intervallen van een vector met noten
 *
 * @author Max
 *
 */
public class Intervals{

    public static Vector<Integer> getIntervals(
    Vector<Integer> PedalPointVector){

        Vector<Integer> Intervallen= new Vector<Integer>(0);

        for(int i= 1; i< PedalPointVector.size(); i++){

            Intervallen.add(PedalPointVector.elementAt(i)-
            PedalPointVector.elementAt(i- 1));
        }

        return Intervallen;
    }

}
```

Bijlage IV.15: Markovizer

```
package pedalpoints;

import java.util.*;

/**
 * Deze class maakt een 1ste orde Markov transitiematrix van de
 * intervallen van een reeks gegeven midinummers.
 *
 * @author Max
 *
 */

public class Markovizer{

    public static HashMap<Integer,HashMap<Integer,Double>>
    makeMarkov(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        Vector<Integer> IntervalElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke intervallen
        for(int i= 0; i< Intervallen.size(); i++){
            if(IntervalElements.contains(Intervallen.elementAt(i))){}else{
                IntervalElements.add(Intervallen.elementAt(i));
            }
        }

        Collections.sort(IntervalElements);

        // Intervalparen worden geteld en in een geneste hashmap
        // gestoken
        int counter= 0;
        int jcounter= 0;
        HashMap<Integer,Integer> Rijtotaal=
        new HashMap<Integer,Integer>();

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< IntervalElements.size(); j++){
                for(int k= 0; k< Intervallen.size()- 1; k++){
                    if(IntervalElements.elementAt(i)== Intervallen
                    .elementAt(k)&&
                    IntervalElements.elementAt(j)== Intervallen
                    .elementAt(k+ 1)){
                        counter++;
                    }
                }// k

                MarkovTrans2.put(IntervalElements.elementAt(j),
                (double) counter);
```

```
                jcounter+= counter;
                counter= 0;
            }// j

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);

            if(jcounter== 0){
                Rijtotaal.put(IntervalElements.elementAt(i), 1);
            }else{
                Rijtotaal
                .put(IntervalElements.elementAt(i), jcounter);
            } // hashmap van rijtotalen om transitiematrix te kunnen
              // maken

            jcounter= 0;
        }// i

        // De hashmap met getelde intervalparen wordt per rij gedeeld
        // door de respectievelijke rijtotalen
        // teneinde een transitiematrix te bekomen
        Double absel;
        double relel= 0;

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< IntervalElements.size(); j++){

                absel=
                MarkovTrans.get(IntervalElements.elementAt(i)).get(
                IntervalElements.elementAt(j));
                relel=
                absel/ Rijtotaal.get(IntervalElements.elementAt(i));
                MarkovTrans2
                .put(IntervalElements.elementAt(j), relel);
            }

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);
        }

        return MarkovTrans;
    }// makeMarkov

}// Markovizer
```

Bijlage IV.16: Markovizer2

```
package pedalpoints;

import java.util.*;

/**
 * Deze class maakt een 2de orde Markov transitiematrix van de
 * intervallen van een reeks gegeven midinummers.
 *
 * @author Max
 *
 */

public class Markovizer2{

    public static
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>
    makeMarkov2(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        Vector<Integer> IntervalElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke intervallen
        for(int i= 0; i< Intervallen.size(); i++){
            if(IntervalElements.contains(Intervallen.elementAt(i))){}else{
                IntervalElements.add(Intervallen.elementAt(i));
            }
        }

        Collections.sort(IntervalElements);

        // Intervalparen worden geteld en in een geneste hashmap
        // gestoken
        int counter= 0;
        int kcounter= 0;
        HashMap<Integer,HashMap<Integer,Integer>> Rijtotaal=
        new HashMap<Integer,HashMap<Integer,Integer>>();

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
            new HashMap<Integer,HashMap<Integer,Double>>();
            HashMap<Integer,Integer> Rijtotaal2=
            new HashMap<Integer,Integer>();

            for(int j= 0; j< IntervalElements.size(); j++){

                HashMap<Integer,Double> MarkovTrans3=
                new HashMap<Integer,Double>();

                for(int k= 0; k< IntervalElements.size(); k++){
                    for(int l= 0; l< Intervallen.size()- 2; l++){

                        if(IntervalElements.elementAt(i)== Intervallen
```

```
                .elementAt(l)&&
                IntervalElements.elementAt(j)== Intervallen
                .elementAt(l+ 1)&&
                IntervalElements.elementAt(k)== Intervallen
                .elementAt(l+ 2)){
                    counter++;
                }
            }// l

            MarkovTrans3.put(IntervalElements.elementAt(k),
            (double) counter);
            kcounter+= counter;
            counter= 0;
        }// k

        MarkovTrans2.put(IntervalElements.elementAt(j),
        MarkovTrans3);

        if(kcounter== 0){
            Rijtotaal2.put(IntervalElements.elementAt(j), 1);
        }else{
            Rijtotaal2.put(IntervalElements.elementAt(j),
            kcounter);
        } // hashmap van rijtotalen om transitiematrix te
          // kunnen maken
        kcounter= 0;
    }// j

    MarkovTrans.put(IntervalElements.elementAt(i),
    MarkovTrans2);
    Rijtotaal.put(IntervalElements.elementAt(i), Rijtotaal2);
}// i

// De hashmap met getelde intervalparen wordt per rij gedeeld
// door de respectievelijke rijtotalen
// teneinde een transitiematrix te bekomen
Double absel;
double relel= 0;

for(int i= 0; i< IntervalElements.size(); i++){

    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
    new HashMap<Integer,HashMap<Integer,Double>>();

    for(int j= 0; j< IntervalElements.size(); j++){

        HashMap<Integer,Double> MarkovTrans3=
        new HashMap<Integer,Double>();

        for(int k= 0; k< IntervalElements.size(); k++){

            absel=
            MarkovTrans.get(IntervalElements.elementAt(i))
            .get(IntervalElements.elementAt(j))
            .get(IntervalElements.elementAt(k));
            relel=
            absel/
            Rijtotaal.get(IntervalElements.elementAt(i)).get(
            IntervalElements.elementAt(j));
```

```
                    MarkovTrans3.put(IntervalElements.elementAt(k),
                    relel);
                }// k

                MarkovTrans2.put(IntervalElements.elementAt(j),
                MarkovTrans3);
            }// j

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);
        }// i

        return MarkovTrans;
    }// makeMarkov

}
```

Bijlage IV.17: PedalPointer

```
package pedalpoints;

import java.util.*;

/**
 *
 * @author Max
 *
 *
 */

public class PedalPointer{

    public PedalPointer(){

        int aantalnoten= 20;
        int aantalrondes= 10;

        // Maken van een eerste oplossing--> vector vullen met
        // willekeurige noten uit E4 Natural Minor
        Vector<Integer> PedalPointVectorE4= new Vector<Integer>(0);

        int[] E4NaturalMinor=
        new int[]{/* E4 */64, 66, 67, 69, 71, 72, 74, 76}; // MIDI
                                                            // nummers

        // PPV vullen
        for(int i= 0; i< aantalnoten; i++){

            int r= E4NaturalMinor.length;
            Random random= new Random();
            int index= random.nextInt(r);
            PedalPointVectorE4.add(E4NaturalMinor[index]);

        }

        // Maken van een Markov transitiematrix vertrekkende van een
        // midisequentie.
        // Bach Toccata & Fugue
        // Hein_1
        // Yngwie Malmsteen Barock&Roll
        // Yngwie Malmsteen Pedalpointing
        int[] MidiSeq=
        new int[]{69, 69, 74, 69, 76, 69, 77, 69, 74, 69, 76, 69, 77,
        69, 79, 69, 76, 69, 77, 69, 79, 69, 81, 69, 77, 69, 79, 69,
        81, 69, 82, 69, 79, 69, 81, 69, 77, 69, 79, 69, 76, 69, 77,
        69, 74, 69, 76, 69, 73, 69, 74, 62, 69, 62, 70, 62, 67, 62,
        69, 62, 65, 62, 67, 62, 64, 62, 65, 57, 62, 57, 67, 57, 64,
        57, 65, 57, 62, 57, 64, 57, 61, 57, 62, 50, 57, 50, 58, 50,
        55, 50, 57, 50, 53, 50, 55, 50, 52, 50, 53, 45, 50, 45, 55,
        45, 52, 45, 53, 45, 50, 45, 52, 45, 49, 45, 50, 73, 69, 73,
        68, 73, 66, 73, 64, 73, 62, 73, 61, 71, 67, 71, 66, 71, 64,
        71, 62, 71, 60, 71, 59, 69, 65, 69, 64, 69, 62, 69, 60, 69,
        58, 69, 57, 67, 63, 67, 62, 67, 60, 67, 58, 67, 56, 67, 55,
        73, 69, 73, 68, 73, 66, 73, 64, 73, 62, 73, 61, 71, 67, 71,
        66, 71, 64, 71, 62, 71, 60, 71, 59, 69, 65, 69, 64, 69, 62,
        69, 60, 69, 58, 69, 57, 67, 63, 67, 62, 67, 60, 67, 58, 67,
        56, 67, 55, 73, 69, 73, 68, 73, 66, 73, 64, 73, 62, 73, 61,
```

```
62, 59, 62, 57, 62, 56, 62, 54, 62, 52, 62, 50, 54, 50, 54,
49, 54, 47, 54, 45, 54, 44, 53, 54, 73, 69, 73, 68, 73, 66,
73, 64, 73, 62, 73, 61, 62, 59, 62, 57, 62, 56, 62, 54, 62,
52, 62, 50, 54, 50, 54, 49, 54, 47, 54, 45, 54, 44, 53, 54,
76, 74, 76, 72, 76, 71, 76, 69, 76, 74, 76, 72, 76, 71, 76,
69, 77, 76, 77, 74, 77, 72, 77, 71, 76, 74, 76, 72, 76, 71,
76, 69, 76, 74, 76, 72, 76, 71, 76, 69, 76, 74, 76, 72, 76,
71, 76, 69, 77, 76, 77, 74, 77, 72, 77, 71, 76, 74, 76, 72,
76, 71, 76, 69, 64, 76, 74, 76, 72, 76, 71, 76, 69, 76, 71,
76, 72, 76, 74, 76, 65, 77, 76, 77, 74, 77, 72, 77, 71, 77,
72, 77, 74, 77, 76, 77, 62, 74, 72, 74, 71, 74, 69, 74, 67,
74, 69, 74, 71, 74, 72, 74, 64, 76, 74, 76, 72, 76, 71, 76,
69, 76, 71, 76, 72, 76, 74, 76, 64, 76, 74, 76, 72, 76, 74,
76, 64, 76, 74, 76, 72, 76, 74, 76, 65, 77, 76, 77, 74, 77,
76, 77, 65, 77, 76, 77, 74, 77, 76, 77, 62, 74, 72, 74, 71,
74, 72, 74, 62, 74, 72, 74, 71, 74, 72, 74, 64, 76, 74, 76,
72, 76, 74, 76, 64, 76, 74, 76, 72, 76, 74, 76, 64, 76, 74,
76, 65, 77, 76, 77, 62, 74, 72, 74, 64, 76, 74, 76, 60, 72,
71, 72, 62, 74, 72, 74, 59, 71, 69, 71, 68, 71, 65, 71, 64,
71, 62, 71, 60, 71, 59, 71, 57};
Vector<Integer> MidiSequentie= new Vector<Integer>(0);

for(int i= 0; i< MidiSeq.length; i++){
    MidiSequentie.add(MidiSeq[i]);
}

// 1ste orde
HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
new HashMap<Integer,HashMap<Integer,Double>>();
MarkovTrans= Markovizer.makeMarkov(MidiSequentie);

// 2de orde
HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2=
new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();
MarkovTrans2= Markovizer2.makeMarkov2(MidiSequentie);

/*
 * //Algoritme 1 PedalPointVectorE4=
 * Algo1LocalSearchFirstDescent0904
 * .berekenAlgo1LocalSearchFirstDescent0904(aantalrondes,
 * aantalnoten, E4NaturalMinor, PedalPointVectorE4);
 */

/*
 * //Algoritme 1v2
 * PedalPointVectorE4=Algo1v2LocalSearchSteepestDescent1004.
 * berekenAlgo1v2LocalSearchSteepestDescent1004(aantalrondes,
 * aantalnoten, E4NaturalMinor, PedalPointVectorE4);
 */

/*
 * //Algoritme 2
 * PedalPointVectorE4=Algo2.berekenAlgo2(aantalrondes,
 * aantalnoten, E4NaturalMinor, PedalPointVectorE4);
 */

/*
 * //Algoritme 2v2
 * PedalPointVectorE4=Algo2v2.berekenAlgo2v2(aantalrondes,
```

```
         * aantalnoten, E4NaturalMinor, PedalPointVectorE4);
         */

        /*
         * //Algoritme 2v2CEM
         * PedalPointVectorE4=Algo2v2CEM.berekenAlgo2v2CEM
         * (aantalrondes, aantalnoten, E4NaturalMinor,
         * PedalPointVectorE4, MarkovTrans);
         */

        /*
         * //Algoritme 1CEM
         * PedalPointVectorE4=Algo1CEM.berekenAlgo1CEM(aantalrondes,
         * aantalnoten, E4NaturalMinor, PedalPointVectorE4,
         * MarkovTrans);
         */

        // Algoritme 2v2CEM2
        PedalPointVectorE4=
        Algo2v2CEM2
        .berekenAlgo2v2CEM2(aantalrondes, aantalnoten,
        E4NaturalMinor, PedalPointVectorE4, MarkovTrans, MarkovTrans2);

        int laatstenoot=
        ZZ_MostCommonPitch.getMostCommonPitch(PedalPointVectorE4);
        PedalPointVectorE4.add(laatstenoot);

        MidiMaker.writeMidiFile(PedalPointVectorE4);
        System.out.println("PedalPointVector: "+ PedalPointVectorE4);

    }// einde PedalPointer()

    // Main
    public static void main(String[] args){
        new PedalPointer();
    }

}// einde class PedalPointer
```

Bijlage IV.18: SmallestElement

```
package pedalpoints;

import java.util.*;

/**
 * Gets the smallest element of a vector of type double
 *
 * @author Max
 *
 */

@SuppressWarnings("unused") public class SmallestElement{

    public static double getSmallest(java.util.Vector<Double> Vector){

        double smallest= 999999999;
        for(int i= 0; i< Vector.size(); i++){

            if(Vector.elementAt(i)< smallest){
                smallest= Vector.elementAt(i);
            }
        }

        return smallest;
    }
}
```

Bijlage IV.19: ZZ_MostCommonPitch

```
package pedalpoints;

import java.util.*;

public class ZZ_MostCommonPitch{

    public static int getMostCommonPitch(
    Vector<Integer> PedalPointVector){

        int most_common_pitch= 0;
        int tempcount5= 0;
        int count5= 0;

        // Berekenen meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))){
                    tempcount5++;
                }
            }
            if(tempcount5> count5){
                most_common_pitch=
                Math.abs(PedalPointVector.elementAt(i));
                count5= tempcount5;
            }
            tempcount5= 0;
        }

        return most_common_pitch;
    }

}
```

Bijlage V: Het project soloGen

Bijlage V.1: Algo2v2CEM2

```
package soloGen;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Hetzelfde als Algo2, maar dan met kleinere stappen teneinde de
 * combinatiemogelijkheden drastisch te beperken. Deze maakt gebruik
 * van een Cross Entropy term in de doelfunctie.
 *
 * De Cross Entropy term wordt berekend op basis van een Markov
 * transitiematrix van de eerste orde en een van de 2de orde.
 *
 * @author Max
 *
 *         27 juni 2014
 *
 */
public class Algo2v2CEM2{

    @SuppressWarnings("unchecked")
    public static
    Vector<Integer>
    berekenAlgo2v2CEM2(
    int aantalrondes,
    int aantalnoten,
    Vector<Integer> MidiElements,
    Vector<Double> jSymVar,
    Vector<Integer> Oplossing,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999999;

        Vector<Integer> Oplossing2=
        new Vector<Integer>(Oplossing.size());

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int r1= MidiElements.size();
        int rp= Oplossing.size();
        Random random1= new Random();

        int nochangecounter= 0;

        int incr= 2;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){
```

```
    Oplossing2= (Vector<Integer>) Oplossing.clone();

    for(int k= 0; k< MidiElements.size(); k++){
        for(int l= 0; l< MidiElements.size(); l++){

            if(j+ 1< aantalnoten){

                Oplossing2.set(j,
                MidiElements.elementAt(k));
                Oplossing2.set(j+ 1,
                MidiElements.elementAt(l));
            }else{
                Oplossing2.set(j,
                MidiElements.elementAt(l));
            }

            doelfunctiewaarde=
            DoelFunctieCEM2.berekenDoelFunctieCEM2(
            Oplossing2, jSymVar, MarkovTrans,
            MarkovTrans2);

            if(doelfunctiewaarde< bestedoelfunctiewaarde){
                bestedoelfunctiewaarde= doelfunctiewaarde;
                Oplossing=
                (Vector<Integer>) Oplossing2.clone();
            }

        }// l
    }// k

    System.out.println("Beste Doelfunctiewaarde: "+
    bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
    ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

}// j

WaardeNaRonde.add(bestedoelfunctiewaarde);

// shaker die willekeurig elementen verandert bij inertie
if(i> 0){
    double verandering=
    WaardeNaRonde.elementAt(i)-
    WaardeNaRonde.elementAt(i- 1);
    if(verandering== 0){

        OplossingenMagazijn1.add(bestedoelfunctiewaarde);
        OplossingenMagazijn2.add(Oplossing);

        for(int n= 0; n< aantalnoten/ 10; n++){

            int index= random1.nextInt(r1);
            int indexp= random1.nextInt(rp);

            Oplossing.set(indexp,
            MidiElements.elementAt(index));
        }

        bestedoelfunctiewaarde=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossing,
```

```
                    jSymVar, MarkovTrans, MarkovTrans2);

                    nochangecounter++;
                }
            }

            // Doet er een aantal rondes bij indien een kritieke
            // waarde niet overschreden wordt.
            if(i== (aantalrondes- 1)){
                System.out.println("bdfw: "+ bestedoelfunctiewaarde);
                if(bestedoelfunctiewaarde< 1.1* SmallestElement
                .getSmallest(OplossingenMagazijn1)){
                    aantalrondes+= 10;
                }
            }

        }// i

        // kijken of de beste oplossing al dan niet de laatste beste
        // oplossing is
        if(SmallestElement.getSmallest(OplossingenMagazijn1)<
             bestedoelfunctiewaarde){

            bestedoelfunctiewaarde=
            SmallestElement.getSmallest(OplossingenMagazijn1);
            int minnie=
            OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
            Oplossing= OplossingenMagazijn2.elementAt(minnie);
        }

        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return Oplossing;
    }// method

}
```

Bijlage V.2: Algo2v2CENM2

```
package soloGen;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Hetzelfde als Algo2, maar dan met kleinere stappen teneinde de
 * combinatiemogelijkheden drastisch te beperken. Deze maakt gebruik
 * van een Cross Entropy term in de doelfunctie.
 *
 * De Cross Entropy term wordt berekend op basis van een Markov
 * transitiematrix van de eerste orde en een van de 2de orde.
 *
 * In tegenstelling tot Algo2v2CEM2 wordt hier een 'noten' TM en geen
 * 'intervallen' TM.
 *
 * @author Max
 *
 *         30 juni 2014
 *
 */
public class Algo2v2CENM2{

    @SuppressWarnings("unchecked")
    public static
    Vector<Integer>
    berekenAlgo2v2CENM2(
    int aantalrondes,
    int aantalnoten,
    Vector<Integer> MidiElements,
    Vector<Double> jSymVar,
    Vector<Integer> Oplossing,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> NotesMarkovTrans2){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999999;

        Vector<Integer> Oplossing2=
        new Vector<Integer>(Oplossing.size());

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);

        int r1= MidiElements.size();
        int rp= Oplossing.size();
        Random random1= new Random();

        int nochangecounter= 0;

        int incr= 1;

        for(int i= 0; i< aantalrondes; i++){

            for(int j= 0; j< aantalnoten; j+= incr){
```

```
    Oplossing2= (Vector<Integer>) Oplossing.clone();

    for(int k= 0; k< MidiElements.size(); k++){
        for(int l= 0; l< MidiElements.size(); l++){

            if(j+ 1< aantalnoten){

                Oplossing2.set(j,
                MidiElements.elementAt(k));
                Oplossing2.set(j+ 1,
                MidiElements.elementAt(l));
            }else{
                Oplossing2.set(j,
                MidiElements.elementAt(l));
            }

            doelfunctiewaarde=
            DoelFunctieCENM2.berekenDoelFunctieCENM2(
            Oplossing2, jSymVar, NotesMarkovTrans,
            NotesMarkovTrans2);

            if(doelfunctiewaarde< bestedoelfunctiewaarde){
                bestedoelfunctiewaarde= doelfunctiewaarde;
                Oplossing=
                (Vector<Integer>) Oplossing2.clone();
            }

        }// l
    }// k

    System.out.println("Beste Doelfunctiewaarde: "+
    bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
    ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

}// j

WaardeNaRonde.add(bestedoelfunctiewaarde);

// shaker die willekeurig elementen verandert bij inertie
if(i> 0){
    double verandering=
    WaardeNaRonde.elementAt(i)-
    WaardeNaRonde.elementAt(i- 1);
    if(verandering== 0){

        OplossingenMagazijn1.add(bestedoelfunctiewaarde);
        OplossingenMagazijn2.add(Oplossing);

        for(int n= 0; n< aantalnoten/ 10; n++){

            int index= random1.nextInt(r1);
            int indexp= random1.nextInt(rp);

            Oplossing.set(indexp,
            MidiElements.elementAt(index));
        }

        bestedoelfunctiewaarde=
```

```
                    DoelFunctieCENM2.berekenDoelFunctieCENM2(
                    Oplossing, jSymVar, NotesMarkovTrans,
                    NotesMarkovTrans2);

                    nochangecounter++;
                }
            }

            // Doet er een aantal rondes bij indien een kritieke
            // waarde niet overschreden wordt.
            if(i== (aantalrondes- 1)){
                System.out.println("bdfw: "+ bestedoelfunctiewaarde);
                if(bestedoelfunctiewaarde< 1.1* SmallestElement
                .getSmallest(OplossingenMagazijn1)){
                    aantalrondes+= 10;
                }
            }
        }// i

        // kijken of de beste oplossing al dan niet de laatste beste
        // oplossing is
        if(SmallestElement.getSmallest(OplossingenMagazijn1)<
             bestedoelfunctiewaarde){

            bestedoelfunctiewaarde=
            SmallestElement.getSmallest(OplossingenMagazijn1);
            int minnie=
            OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
            Oplossing= OplossingenMagazijn2.elementAt(minnie);
        }

        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return Oplossing;
    }// method

}
```

Bijlage V.3: Algo3

```
package soloGen;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Bouwt verder op Algo2v2 en variaties. Ipv neighbours te zoeken door
 * telkens alle variaties van 2 noten te beschouwen, worden er hier
 * neighbours gegenereerd adhv van Markov Chains.
 *
 * @author Max
 *
 */

public class Algo3{

    @SuppressWarnings("unchecked")
    public static
    Vector<Integer>
    berekenAlgo3(
    int aantalrondes,
    int aantalnoten,
    Vector<Integer> MidiElements,
    Vector<Double> jSymVar,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans,
    Vector<Integer> Oplossing,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2){

        double doelfunctiewaarde= 0;
        double bestedoelfunctiewaarde= 999999999;

        Vector<Integer> Oplossing2=
        new Vector<Integer>(Oplossing.size());

        Vector<Integer> NB1= new Vector<Integer>(0);
        Vector<Integer> NB2= new Vector<Integer>(0);
        Vector<Integer> NB3= new Vector<Integer>(0);
        Vector<Integer> Oplossingx1=
        new Vector<Integer>(Oplossing.size());
        Vector<Integer> Oplossingx2=
        new Vector<Integer>(Oplossing.size());
        Vector<Integer> Oplossingx3=
        new Vector<Integer>(Oplossing.size());
        Vector<Integer> pert= new Vector<Integer>(0);
        double dfwaarde1= 0;
        double dfwaarde2= 0;
        double dfwaarde3= 0;
        Vector<Vector<Integer>> TabuList=
        new Vector<Vector<Integer>>(0);

        int firstnote= 0;

        Vector<Double> WaardeNaRonde= new Vector<Double>(0);
        Vector<Double> OplossingenMagazijn1= new Vector<Double>(0);
        Vector<Vector<Integer>> OplossingenMagazijn2=
        new Vector<Vector<Integer>>(0);
```

```
// int r1= MidiElements.size();
int rp= Oplossing.size();
Random random1= new Random();
int indx= 0;

int nochangecounter= 0;

int incr= (aantalnoten/ 20)+ 1;

for(int i= 0; i< aantalrondes; i++){

    for(int j= 0; j< aantalnoten- incr; j+= incr){

        Oplossing2= (Vector<Integer>) Oplossing.clone();
        Oplossingx1= (Vector<Integer>) Oplossing.clone();
        Oplossingx2= (Vector<Integer>) Oplossing.clone();
        Oplossingx3= (Vector<Integer>) Oplossing.clone();

        if(j== 0){
            firstnote= MidiElements.elementAt(indx);
        }else{
            firstnote= Oplossing2.elementAt(j- 1);
        }

        do{
            // indx= random1.nextInt(r1);
            // firstnote= MidiElements.elementAt(indx);
            NB1=
            MarkovChainer.makeChain(incr, firstnote,
            MidiElements, NotesMarkovTrans);

        }while(TabuList.contains(NB1)|| NB1.contains(0));

        do{
            // indx= random1.nextInt(r1);
            // firstnote= MidiElements.elementAt(indx);
            NB2=
            MarkovChainer.makeChain(incr, firstnote,
            MidiElements, NotesMarkovTrans);

        }while(TabuList.contains(NB2)|| NB2.contains(0));

        do{
            // indx= random1.nextInt(r1);
            // firstnote= MidiElements.elementAt(indx);
            NB3=
            MarkovChainer.makeChain(incr, firstnote,
            MidiElements, NotesMarkovTrans);

        }while(TabuList.contains(NB3)|| NB3.contains(0));

        for(int k= 0; k< incr; k++){

            Oplossingx1.set(j+ k, NB1.elementAt(k));
            Oplossingx2.set(j+ k, NB2.elementAt(k));
            Oplossingx3.set(j+ k, NB3.elementAt(k));
        }
```

```
        dfwaarde1=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossingx1,
        jSymVar, MarkovTrans, MarkovTrans2);
        dfwaarde2=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossingx2,
        jSymVar, MarkovTrans, MarkovTrans2);
        dfwaarde3=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossingx3,
        jSymVar, MarkovTrans, MarkovTrans2);

        if(dfwaarde1< dfwaarde2&& dfwaarde1< dfwaarde3){

            for(int k= 0; k< incr; k++){
                Oplossing2.set(j+ k, NB1.elementAt(k));
            }
            TabuList.add(NB1);
            // TabuList.add(NB2);
            // TabuList.add(NB3);
        }else if(dfwaarde2< dfwaarde1&& dfwaarde2< dfwaarde3){

            for(int k= 0; k< incr; k++){
                Oplossing2.set(j+ k, NB2.elementAt(k));
            }
            // TabuList.add(NB1);
            TabuList.add(NB2);
            // TabuList.add(NB3);
        }else{

            for(int k= 0; k< incr; k++){
                Oplossing2.set(j+ k, NB3.elementAt(k));
            }
            // TabuList.add(NB1);
            // TabuList.add(NB2);
            TabuList.add(NB3);
        }

        doelfunctiewaarde=
        DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossing2,
        jSymVar, MarkovTrans, MarkovTrans2);

        if(doelfunctiewaarde< bestedoelfunctiewaarde){
            bestedoelfunctiewaarde= doelfunctiewaarde;
            Oplossing= (Vector<Integer>) Oplossing2.clone();

        }

        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde+ " Ronde: "+ (i+ 1)+
        ", noten "+ (j+ 1)+ " t.e.m "+ (j+ incr));

    }// j

    WaardeNaRonde.add(bestedoelfunctiewaarde);

    // shaker die elementen verandert bij inertie
    if(i> 0.1* aantalnoten){
        double verandering=
        WaardeNaRonde.elementAt(i)-
        WaardeNaRonde.elementAt((int) (i- 0.1* aantalnoten));
```

```
        if(verandering== 0){

            OplossingenMagazijn1.add(bestedoelfunctiewaarde);
            OplossingenMagazijn2.add(Oplossing);

            for(int n= 0; n< aantalnoten/ 10; n++){

                // int index = random1.nextInt(r1);
                int indexp= Math.max(1, random1.nextInt(rp));

                // Oplossing.set(indexp,MidiElements.elementAt(index));
                pert=
                MarkovChainer.makeChain(2,
                Oplossing.get(indexp- 1), MidiElements,
                NotesMarkovTrans);
                Oplossing.set(indexp, pert.elementAt(1));
                pert.clear();

            }

            bestedoelfunctiewaarde=
            DoelFunctieCEM2.berekenDoelFunctieCEM2(Oplossing,
            jSymVar, MarkovTrans, MarkovTrans2);

            nochangecounter++;
        }
    }

    // Doet er een aantal rondes bij indien een kritieke
    // waarde niet overschreden wordt.
    if(i== (aantalrondes- 1)){
        if(bestedoelfunctiewaarde< 1.1* SmallestElement
        .getSmallest(OplossingenMagazijn1) /*
                                           * ||
                                           * bestedoelfunctiewaarde
                                           * >
                                           * (0.8*aantalnoten
                                           * )
                                           */){

            if(nochangecounter> 3|| aantalrondes> 1000){}else{
                aantalrondes+= 10;
            }
        }
    }

    TabuList.remove(0);
}// i

// kijken of de beste oplossing al dan niet de laatste beste
// oplossing is
if(SmallestElement.getSmallest(OplossingenMagazijn1)<
     bestedoelfunctiewaarde){
    bestedoelfunctiewaarde=
    SmallestElement.getSmallest(OplossingenMagazijn1);
    int minnie=
    OplossingenMagazijn1.indexOf(bestedoelfunctiewaarde);
    Oplossing= OplossingenMagazijn2.elementAt(minnie);
}
```

```
        System.out.println("aantal shakes: "+ nochangecounter);
        System.out.println("Optima: "+ OplossingenMagazijn1);
        System.out.println("Beste Doelfunctiewaarde: "+
        bestedoelfunctiewaarde);

        return Oplossing;
    }// Method
}// Class
```

Bijlage V.4: DoelFunctieCEM2

```
package soloGen;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * Bijkomend is er in de doelfunctie een soort van Cross Entropy term
 * gemaakt die een fout toevoegd gebaseerd op een Markov
 * transitiematrix.
 *
 * In deze doelfunctie is een Markov transitiematrix van zowel de
 * eerste als de tweede orde verwerkt.
 *
 * @author Max
 *
 */

public class DoelFunctieCEM2{

    public static
    double
    berekenDoelFunctieCEM2(
    Vector<Integer> Oplossing,
    Vector<Double> jSymVar,
    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2){

        // jSymbolic variabelen van de Oplossing berekenen
        Vector<Double> OplVar= new Vector<Double>(0);
        OplVar= JSymVar.getJSymVar(Oplossing);

        // Berekenen van de procentuele afwijkingen op de jSymbolic
        // gebaseerde referentiewaarden.
        double doelfunctiewaarde= 0;

        for(int i= 0; i< OplVar.size(); i++){

            if(jSymVar.elementAt(i)!= 0){
                doelfunctiewaarde+=
                (double) Math
                .abs((double) (OplVar.elementAt(i)- jSymVar
                .elementAt(i))/ jSymVar.elementAt(i));
            }
        }

        doelfunctiewaarde= 0.2* doelfunctiewaarde;

        // Cross Entropy fout toevoegen aan doelfunctie.

        // Markov 1ste orde
        double TransProb= 0;
        double CEfout= 0;
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();
        HashMap<Integer,Double> InnerMapx2=
```

```
        new HashMap<Integer,Double>();
        HashMap<Integer,HashMap<Integer,Double>> InnerMapx=
        new HashMap<Integer,HashMap<Integer,Double>>();
        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(Oplossing);

        for(int i= 0; i< Intervallen.size()- 1; i++){

            try{
                InnerMap= MarkovTrans.get(Intervallen.elementAt(i));
                TransProb= InnerMap.get(Intervallen.elementAt(i+ 1));
            }catch(NullPointerException e){
                TransProb= 0;
            }// Een catch voor het geval een intervalpaar niet in de
             // transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb< 0.000001){
                CEfout= 6;
            }else{
                CEfout= Math.log10(1/ TransProb);
            }

            doelfunctiewaarde+= 0.4* CEfout;
        }

        // Markov 2de orde
        double TransProb2= 0;
        double CEfout2= 0;

        for(int i= 0; i< Intervallen.size()- 2; i++){

            try{
                InnerMapx= MarkovTrans2.get(Intervallen.elementAt(i));
                InnerMapx2=
                InnerMapx.get(Intervallen.elementAt(i+ 1));
                TransProb2=
                InnerMapx2.get(Intervallen.elementAt(i+ 2));
            }catch(NullPointerException e){
                TransProb2= 0;
            }// Een catch voor het geval een intervaltriplet niet in
             // de transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb2< 0.000001){
                CEfout2= 6;
            }else{
                CEfout2= Math.log10(1/ TransProb2);
            }

            doelfunctiewaarde+= 0.4* CEfout2;
        }

        return doelfunctiewaarde;

    }// einde method

}
```

Bijlage V.5: DoelFunctieCENM2

```
package soloGen;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * Bijkomend is er in de doelfunctie een soort van Cross Entropy term
 * gemaakt die een fout toevoegd gebaseerd op een Markov
 * transitiematrix.
 *
 * In deze doelfunctie is een Markov transitiematrix van zowel de
 * eerste als de tweede orde verwerkt.
 *
 * Het verschil met DoelFunctieCEM2 is dat deze gebruik maakt van een
 * noten Markov ipv intervallen Markov transitiematrix.
 *
 * @author Max
 *
 */

public class DoelFunctieCENM2{

    public static
    double
    berekenDoelFunctieCENM2(
    Vector<Integer> Oplossing,
    Vector<Double> jSymVar,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans,
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> NotesMarkovTrans2){

        // jSymbolic variabelen van de Oplossing berekenen
        Vector<Double> OplVar= new Vector<Double>(0);
        OplVar= JSymVar.getJSymVar(Oplossing);

        // Berekenen van de procentuele afwijkingen op de jSymbolic
        // gebaseerde referentiewaarden.
        double doelfunctiewaarde= 0;
        for(int i= 0; i< OplVar.size(); i++){

            if(jSymVar.elementAt(i)!= 0){
                doelfunctiewaarde+=
                (double) Math
                .abs((double) (OplVar.elementAt(i)- jSymVar
                .elementAt(i))/ jSymVar.elementAt(i));
            }
        }

        // Cross Entropy fout toevoegen aan doelfunctie.

        // Markov 1ste orde
        double TransProb= 0;
        double CEfout= 0;

        for(int i= 0; i< Oplossing.size()- 1; i++){
```

```
            try{
                TransProb=
                NotesMarkovTrans.get(Oplossing.elementAt(i)).get(
                Oplossing.elementAt(i+ 1));
            }catch(NullPointerException e){
                TransProb= 0;
            }// Een catch voor het geval een notenpaar niet in de
             // transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb< 0.000001){
                CEfout= 6;
            }else{
                CEfout= Math.log10(1/ TransProb);
            }

            doelfunctiewaarde+= CEfout;
        }

        // Markov 2de orde
        double TransProb2= 0;
        double CEfout2= 0;

        for(int i= 0; i< Oplossing.size()- 2; i++){

            try{
                TransProb2=
                NotesMarkovTrans2.get(Oplossing.elementAt(i))
                .get(Oplossing.elementAt(i+ 1))
                .get(Oplossing.elementAt(i+ 2));
            }catch(NullPointerException e){
                TransProb2= 0;
            }// Een catch voor het geval een notentriplet niet in de
             // transitiematrix aanwezig is.

            // lim x-->0 log(1/x) = oneindig vermijden!
            if(TransProb2< 0.000001){
                CEfout2= 6;
            }else{
                CEfout2= Math.log10(1/ TransProb2);
            }

            doelfunctiewaarde+= CEfout2;
        }

        return doelfunctiewaarde;

    }// einde method

}// einde class
```

Bijlage V.6: Intervals

```
package soloGen;

import java.util.*;

/**
 * berekent de vector met intervallen van een vector met noten
 *
 * @author Max
 *
 */
public class Intervals{

    public static Vector<Integer> getIntervals(
    Vector<Integer> PedalPointVector){

        Vector<Integer> Intervallen= new Vector<Integer>(0);

        for(int i= 1; i< PedalPointVector.size(); i++){

            Intervallen.add(PedalPointVector.elementAt(i)-
            PedalPointVector.elementAt(i- 1));
        }

        return Intervallen;
    }

}
```

Bijlage V.7: JSymVar

```
package soloGen;

import java.util.Vector;

/**
 *
 * @author Max
 *
 */

public class JSymVar{

    public static Vector<Double> getJSymVar(
    Vector<Integer> MidiSequentie){

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        // Vector maken voor waarden van berekende variabelen
        // (jSymbolic gebaseerd)
        Vector<Double> jSymVar= new Vector<Double>(0);

        double arpeg=
        AmountOfArpeggiation.getAmountOfArpeggiation(Intervallen);
        jSymVar.add(arpeg);

        double avmi=
        AverageMelodicInterval.getAverageMelodicInterval(Intervallen);
        jSymVar.add(avmi);

        double chromes=
        ChromaticMotion.getChromaticMotion(Intervallen);
        jSymVar.add(chromes);

        double dm=
        DirectionOfMotion.getDirectionOfMotion(Intervallen);
        jSymVar.add(dm);

        double discommonint=
        DistanceBetweenMostCommonMelodicIntervals
        .getDistanceBetweenMostCommonMelodicIntervals(Intervallen);
        jSymVar.add(discommonint);

        double doma=
        DurationOfMelodicArcs.getDurationOfMelodicArcs(MidiSequentie,
        Intervallen);
        jSymVar.add(doma);

        double inbesp=
        IntervalBetweenStrongestPitches
        .getIntervalBetweenStrongestPitches(MidiSequentie);
        jSymVar.add(inbesp);

        double fifthz= MelodicFifths.getMelodicFifths(Intervallen);
        jSymVar.add(fifthz);

        double octavez= MelodicOctaves.getMelodicOctaves(Intervallen);
        jSymVar.add(octavez);
```

```
        double thirdz= MelodicThirds.getMelodicThirds(Intervallen);
        jSymVar.add(thirdz);

        double tritonez=
        MelodicTritones.getMelodicTritones(Intervallen);
        jSymVar.add(tritonez);

        double commelint=
        MostCommonMelodicInterval
        .getMostCommonMelodicInterval(Intervallen);
        jSymVar.add(commelint);

        double commelintprev=
        MostCommonMelodicIntervalPrevalence
        .getMostCommonMelodicIntervalPrevalence(Intervallen);
        jSymVar.add(commelintprev);

        double compitprev=
        MostCommonPitchPrevalence
        .getMostCommonPitchPrevalence(MidiSequentie);
        jSymVar.add(compitprev);

        double ncommint=
        NumberOfCommonMelodicIntervals
        .getNumberOfCommonMelodicIntervals(Intervallen);
        jSymVar.add(ncommint);

        double numcompit=
        NumberOfCommonPitches.getNumberOfCommonPitches(MidiSequentie);
        jSymVar.add(numcompit);

        double relstrcommint=
        RelativeStrengthOfMostCommonIntervals
        .getRelativeStrengthOfMostCommonIntervals(Intervallen);
        jSymVar.add(relstrcommint);

        double relstrtopitch=
        RelativeStrengthOfTopPitch
        .getRelativeStrengthOfTopPitch(MidiSequentie);
        jSymVar.add(relstrtopitch);

        double repnote= RepeatedNotes.getRepeatedNotes(Intervallen);
        jSymVar.add(repnote);

        double sizarcs=
        SizeOfMelodicArcs.getSizeOfMelodicArcs(Intervallen);
        jSymVar.add(sizarcs);

        double stepmo= StepwiseMotion.getStepwiseMotion(Intervallen);
        jSymVar.add(stepmo);

        return jSymVar;
    }

}
```

Bijlage V.8: MarkovChainer

```
package soloGen;

import java.util.HashMap;
import java.util.Vector;

/**
 * Maakt een Markov Chain gegeven een transitiematrix.
 *
 * @author Max
 *
 */

public class MarkovChainer{

    public static Vector<Integer> makeChain(int aantalnoten,
    int firstnote, Vector<Integer> MidiElements,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans){

        Vector<Integer> MarkovChain= new Vector<Integer>(aantalnoten);
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();
        double P= 0;
        double rand= 0;
        int nieuwelement= 0;

        for(int i= 0; i< aantalnoten; i++){
            if(i== 0){

                MarkovChain.add(firstnote);
            }else{

                do{
                    rand= Math.random();
                }while(rand< 0.001);

                P= 0;

                for(int j= 0; j< MidiElements.size()+ 1; j++){


                    if(P< rand){

                        InnerMap=
                        NotesMarkovTrans.get(MarkovChain
                        .elementAt(i- 1));
                        P+= InnerMap.get(MidiElements.elementAt(j));
                    }else{
                        nieuwelement= MidiElements.elementAt(j- 1);
                        break;
                    }
                }
                MarkovChain.add(nieuwelement);
            }
        }
        return MarkovChain;
    }

}
```

Bijlage V.9: Markovizer

```
package soloGen;

import java.util.*;

/**
 * Deze class maakt een 1ste orde Markov transitiematrix van de
 * intervallen van een reeks gegeven midinummers.
 *
 * @author Max
 *
 */

public class Markovizer{

    public static HashMap<Integer,HashMap<Integer,Double>>
    makeMarkov(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        Vector<Integer> IntervalElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke intervallen
        for(int i= 0; i< Intervallen.size(); i++){
            if(IntervalElements.contains(Intervallen.elementAt(i))){}else{
                IntervalElements.add(Intervallen.elementAt(i));
            }
        }

        Collections.sort(IntervalElements);

        // Intervalparen worden geteld en in een geneste hashmap
        // gestoken
        int counter= 0;
        int jcounter= 0;
        HashMap<Integer,Integer> Rijtotaal=
        new HashMap<Integer,Integer>();

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< IntervalElements.size(); j++){
                for(int k= 0; k< Intervallen.size()- 1; k++){
                    if(IntervalElements.elementAt(i)== Intervallen
                    .elementAt(k)&&
                    IntervalElements.elementAt(j)== Intervallen
                    .elementAt(k+ 1)){
                        counter++;
                    }
                }// k

                MarkovTrans2.put(IntervalElements.elementAt(j),
                (double) counter);
```

```
                jcounter+= counter;
                counter= 0;
            }// j

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);

            if(jcounter== 0){
                Rijtotaal.put(IntervalElements.elementAt(i), 1);
            }else{
                Rijtotaal
                .put(IntervalElements.elementAt(i), jcounter);
            }// hashmap van rijtotalen om transitiematrix te kunnen
             // maken

            jcounter= 0;
        }// i

        // De hashmap met getelde intervalparen wordt per rij gedeeld
        // door de respectievelijke rijtotalen
        // teneinde een transitiematrix te bekomen
        Double absel;
        double relel= 0;
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< IntervalElements.size(); j++){

                InnerMap=
                MarkovTrans.get(IntervalElements.elementAt(i));
                absel= InnerMap.get(IntervalElements.elementAt(j));
                relel=
                absel/ Rijtotaal.get(IntervalElements.elementAt(i));
                MarkovTrans2
                .put(IntervalElements.elementAt(j), relel);
            }

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);
        }

        return MarkovTrans;
    }// makeMarkov

}// Markovizer
```

Bijlage V.10: Markovizer2

```
package soloGen;

import java.util.*;

/**
 * Deze class maakt een 2de orde Markov transitiematrix van de
 * intervallen van een reeks gegeven midinummers.
 *
 * @author Max
 *
 */

public class Markovizer2{

    public static
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>
    makeMarkov2(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        Vector<Integer> IntervalElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke intervallen
        for(int i= 0; i< Intervallen.size(); i++){
            if(IntervalElements.contains(Intervallen.elementAt(i))){}else{
                IntervalElements.add(Intervallen.elementAt(i));
            }
        }

        Collections.sort(IntervalElements);

        // Intervalparen worden geteld en in een geneste hashmap
        // gestoken
        int counter= 0;
        int kcounter= 0;
        HashMap<Integer,HashMap<Integer,Integer>> Rijtotaal=
        new HashMap<Integer,HashMap<Integer,Integer>>();

        for(int i= 0; i< IntervalElements.size(); i++){

            HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
            new HashMap<Integer,HashMap<Integer,Double>>();
            HashMap<Integer,Integer> Rijtotaal2=
            new HashMap<Integer,Integer>();

            for(int j= 0; j< IntervalElements.size(); j++){

                HashMap<Integer,Double> MarkovTrans3=
                new HashMap<Integer,Double>();

                for(int k= 0; k< IntervalElements.size(); k++){
                    for(int l= 0; l< Intervallen.size()- 2; l++){

                        if(IntervalElements.elementAt(i)== Intervallen
```

```
                .elementAt(l)&&
                IntervalElements.elementAt(j)== Intervallen
                .elementAt(l+ 1)&&
                IntervalElements.elementAt(k)== Intervallen
                .elementAt(l+ 2)){
                    counter++;
                }
            }// l

            MarkovTrans3.put(IntervalElements.elementAt(k),
            (double) counter);
            kcounter+= counter;
            counter= 0;
        }// k

        MarkovTrans2.put(IntervalElements.elementAt(j),
        MarkovTrans3);

        if(kcounter== 0){
            Rijtotaal2.put(IntervalElements.elementAt(j), 1);
        }else{
            Rijtotaal2.put(IntervalElements.elementAt(j),
            kcounter);
        } // hashmap van rijtotalen om transitiematrix te
          // kunnen maken
        kcounter= 0;
    }// j

    MarkovTrans.put(IntervalElements.elementAt(i),
    MarkovTrans2);
    Rijtotaal.put(IntervalElements.elementAt(i), Rijtotaal2);
}// i

// De hashmap met getelde intervalparen wordt per rij gedeeld
// door de respectievelijke rijtotalen
// teneinde een transitiematrix te bekomen
Double absel;
double relel= 0;
HashMap<Integer,HashMap<Integer,Double>> InnerMap=
new HashMap<Integer,HashMap<Integer,Double>>();
HashMap<Integer,Double> InnerMap2=
new HashMap<Integer,Double>();
HashMap<Integer,Integer> Rijtotaalx=
new HashMap<Integer,Integer>();

for(int i= 0; i< IntervalElements.size(); i++){

    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
    new HashMap<Integer,HashMap<Integer,Double>>();

    for(int j= 0; j< IntervalElements.size(); j++){

        HashMap<Integer,Double> MarkovTrans3=
        new HashMap<Integer,Double>();

        for(int k= 0; k< IntervalElements.size(); k++){

            InnerMap=
            MarkovTrans.get(IntervalElements.elementAt(i));
```

```
                    InnerMap2=
                    InnerMap.get(IntervalElements.elementAt(j));
                    absel=
                    InnerMap2.get(IntervalElements.elementAt(k));

                    Rijtotaalx=
                    Rijtotaal.get(IntervalElements.elementAt(i));
                    relel=
                    absel/
                    Rijtotaalx.get(IntervalElements.elementAt(j));
                    MarkovTrans3.put(IntervalElements.elementAt(k),
                    relel);
                }// k

                MarkovTrans2.put(IntervalElements.elementAt(j),
                MarkovTrans3);
            }// j

            MarkovTrans.put(IntervalElements.elementAt(i),
            MarkovTrans2);
        }// i

        return MarkovTrans;
    }// makeMarkov

}
```

Bijlage V.11: NotesMarkovizer

```
package soloGen;

import java.util.Collections;
import java.util.HashMap;
import java.util.Vector;

/**
 * Deze class maakt een Markov TransitieMatrix van een reeks gegeven
 * midinummers.
 *
 * @author Max
 *
 */

public class NotesMarkovizer{

    public static HashMap<Integer,HashMap<Integer,Double>>
    makeMarkov(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();

        Vector<Integer> MidiElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke noten
        for(int i= 0; i< MidiSequentie.size(); i++){
            if(MidiElements.contains(MidiSequentie.elementAt(i))){}else{
                MidiElements.add(MidiSequentie.elementAt(i));
            }
        }

        Collections.sort(MidiElements);

        // Nootparen worden geteld en in een geneste hashmap gestoken
        int counter= 0;
        int jcounter= 0;
        HashMap<Integer,Integer> Rijtotaal=
        new HashMap<Integer,Integer>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){
                for(int k= 0; k< MidiSequentie.size()- 1; k++){
                    if(MidiElements.elementAt(i)== MidiSequentie
                    .elementAt(k)&&
                    MidiElements.elementAt(j)== MidiSequentie
                    .elementAt(k+ 1)){
                        counter++;
                    }
                }// k

                MarkovTrans2.put(MidiElements.elementAt(j),
                (double) counter);
                jcounter+= counter;
                counter= 0;
```

```
            }// j

            MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);

            if(jcounter== 0){
                Rijtotaal.put(MidiElements.elementAt(i), 1);
            }else{
                Rijtotaal.put(MidiElements.elementAt(i), jcounter);
            }// hashmap van rijtotalen om transitiematrix te kunnen
             // maken

            jcounter= 0;
        }// i

        // De hashmap met getelde nootparen wordt per rij gedeeld door
        // de respectievelijke rijtotalen
        // teneinde een transitiematrix te bekomen
        Double absel;
        double relel= 0;
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){

                InnerMap= MarkovTrans.get(MidiElements.elementAt(i));
                absel= InnerMap.get(MidiElements.elementAt(j));
                relel=
                absel/ Rijtotaal.get(MidiElements.elementAt(i));
                MarkovTrans2.put(MidiElements.elementAt(j), relel);
            }

            MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);
        }

        return MarkovTrans;
    }// makeMarkov

}
```

Bijlage V.12: NotesMarkovizer2

```
package soloGen;

import java.util.*;

/**
 * Deze class maakt een 2de orde Markov transitiematrix van een reeks
 * gegeven midinummers.
 *
 * @author Max
 *
 */

public class NotesMarkovizer2{

    public static
    HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>
    makeMarkov2(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans=
        new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();

        Vector<Integer> MidiElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke noten
        for(int i= 0; i< MidiSequentie.size(); i++){
            if(MidiElements.contains(MidiSequentie.elementAt(i))){}else{
                MidiElements.add(MidiSequentie.elementAt(i));
            }
        }

        Collections.sort(MidiElements);

        // Intervalparen worden geteld en in een geneste hashmap
        // gestoken
        int counter= 0;
        int kcounter= 0;
        HashMap<Integer,HashMap<Integer,Integer>> Rijtotaal=
        new HashMap<Integer,HashMap<Integer,Integer>>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
            new HashMap<Integer,HashMap<Integer,Double>>();
            HashMap<Integer,Integer> Rijtotaal2=
            new HashMap<Integer,Integer>();

            for(int j= 0; j< MidiElements.size(); j++){

                HashMap<Integer,Double> MarkovTrans3=
                new HashMap<Integer,Double>();

                for(int k= 0; k< MidiElements.size(); k++){
                    for(int l= 0; l< MidiSequentie.size()- 2; l++){

                        if(MidiElements.elementAt(i)== MidiSequentie
                        .elementAt(l)&&
                        MidiElements.elementAt(j)== MidiSequentie
                        .elementAt(l+ 1)&&
```

```
                MidiElements.elementAt(k)== MidiSequentie
                .elementAt(l+ 2)){
                    counter++;
                }
            }// l

            MarkovTrans3.put(MidiElements.elementAt(k),
            (double) counter);
            kcounter+= counter;
            counter= 0;
        }// k

        MarkovTrans2.put(MidiElements.elementAt(j),
        MarkovTrans3);

        if(kcounter== 0){
            Rijtotaal2.put(MidiElements.elementAt(j), 1);
        }else{
            Rijtotaal2.put(MidiElements.elementAt(j),
            kcounter);
        } // hashmap van rijtotalen om transitiematrix te
          // kunnen maken
        kcounter= 0;
    }// j

    MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);
    Rijtotaal.put(MidiElements.elementAt(i), Rijtotaal2);
}// i

// De hashmap met getelde nootparen wordt per rij gedeeld door
// de respectievelijke rijtotalen
// teneinde een transitiematrix te bekomen
Double absel;
double relel= 0;

for(int i= 0; i< MidiElements.size(); i++){

    HashMap<Integer,HashMap<Integer,Double>> MarkovTrans2=
    new HashMap<Integer,HashMap<Integer,Double>>();

    for(int j= 0; j< MidiElements.size(); j++){

        HashMap<Integer,Double> MarkovTrans3=
        new HashMap<Integer,Double>();

        for(int k= 0; k< MidiElements.size(); k++){

            absel=
            MarkovTrans.get(MidiElements.elementAt(i))
            .get(MidiElements.elementAt(j))
            .get(MidiElements.elementAt(k));
            relel=
            absel/
            Rijtotaal.get(MidiElements.elementAt(i)).get(
            MidiElements.elementAt(j));
            MarkovTrans3
            .put(MidiElements.elementAt(k), relel);
        }// k
```

```
                MarkovTrans2.put(MidiElements.elementAt(j),
                MarkovTrans3);
            }// j

            MarkovTrans.put(MidiElements.elementAt(i), MarkovTrans2);
        }// i

        return MarkovTrans;
    }// makeMarkov

}
```

Bijlage V.13: SmallestElement

```
package soloGen;

import java.util.*;

/**
 * Gets the smallest element of a vector of type double
 *
 * @author Max
 *
 */

@SuppressWarnings("unused") public class SmallestElement{

    public static double getSmallest(java.util.Vector<Double> Vector){

        double smallest= 999999999;
        for(int i= 0; i< Vector.size(); i++){

            if(Vector.elementAt(i)< smallest){
                smallest= Vector.elementAt(i);
            }
        }

        return smallest;
    }
}
```

Bijlage V.14: SoloGenMain

```
package soloGen;

import java.util.Collections;
import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 *
 * @author Max
 *
 */

public class SoloGenMain{

    @SuppressWarnings("unused")
    public SoloGenMain(){

        // Referentiesolo Dream Theater - Build Me Up, Break Me Down
        // @4min20
        int[] MidiSeq=
        new int[]{76, 80, 76, 73, 68, 64, 61, 56, 61, 64, 68, 73, 76,
        80, 76, 73, 68, 73, 76, 85, 80, 76, 73, 76, 80, 85, 80, 84,
        80, 81, 80, 78, 76, 75, 73, 72, 69, 66, 69, 72, 75, 78, 81,
        78, 81, 78, 75, 72, 69, 72, 75, 78, 81, 78, 84, 80, 81, 80,
        78, 80, 76, 78, 75, 76, 75, 73, 72, 68, 72, 75, 78, 76, 73,
        68, 64, 61, 64, 68, 73, 76, 73, 80, 73, 81, 73, 80, 73, 85,
        80, 76, 73, 68, 64, 68, 73, 76, 80, 85, 80, 76, 80, 73, 80,
        81, 78, 75, 72, 69, 72, 66, 69, 63, 66, 60, 63, 57, 60, 63,
        66, 69, 72, 75, 72, 68, 66, 68, 72, 75, 72, 68, 72, 75, 78,
        80, 84, 80, 78, 75, 78, 84, 87, 84, 80, 75, 80, 78};

        int aantalnoten= MidiSeq.length;
        int aantalrondes= 10;

        Vector<Integer> MidiSequentie= new Vector<Integer>(0);
        Vector<Integer> MidiElements= new Vector<Integer>(0);

        for(int i= 0; i< MidiSeq.length; i++){
            MidiSequentie.add(MidiSeq[i]);
        }

        // Een vector vullen met de unieke elementen uit MidiSeq
        for(int i= 0; i< MidiSeq.length; i++){
            if(MidiElements.contains(MidiSeq[i])){}else{
                MidiElements.add(MidiSeq[i]);
            }
        }
        System.out.println("MidiElements: "+ MidiElements);

        Collections.sort(MidiElements);

        // jSymVar
        Vector<Double> jSymVar= new Vector<Double>(0);
        jSymVar= JSymVar.getJSymVar(MidiSequentie);
        System.out.println("jSymVar: "+ jSymVar);

        // 1ste orde transitiematrix van de noten
```

```
HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans=
new HashMap<Integer,HashMap<Integer,Double>>();
NotesMarkovTrans= NotesMarkovizer.makeMarkov(MidiSequentie);
System.out.println("NotesMarkovTrans: "+ NotesMarkovTrans);

// 2de orde transitiematrix van de noten
HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>
 NotesMarkovTrans2=
new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();
NotesMarkovTrans2=
NotesMarkovizer2.makeMarkov2(MidiSequentie);

// 1ste orde transitiematrix van de intervallen
HashMap<Integer,HashMap<Integer,Double>> MarkovTrans=
new HashMap<Integer,HashMap<Integer,Double>>();
MarkovTrans= Markovizer.makeMarkov(MidiSequentie);

// 2de orde transitiematrix van de intervallen
HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>> MarkovTrans2=
new HashMap<Integer,HashMap<Integer,HashMap<Integer,Double>>>();
MarkovTrans2= Markovizer2.makeMarkov2(MidiSequentie);

/*
 * //Maken van een eerste willekeurige oplossing
 * Vector<Integer> Oplossing = new Vector<Integer>(0);
 *
 * for(int i=0; i<aantalnoten; i++){
 *
 * int r = MidiElements.size(); Random random = new Random();
 * int index = random.nextInt(r);
 * Oplossing.add(MidiElements.elementAt(index)); }
 * System.out.println("Oplossing: "+Oplossing);
 */

// Maak een eerste oplossing gebaseerd op Markov 1ste orde TM
// (noten, niet intervallen)
int firstnote= 0;
int r= MidiElements.size();
Random random= new Random();
int index= random.nextInt(r);
firstnote= MidiElements.elementAt(index);

Vector<Integer> Oplossing= new Vector<Integer>(aantalnoten);
Oplossing=
MarkovChainer.makeChain(aantalnoten, firstnote, MidiElements,
NotesMarkovTrans);

// Algoritme aanroepen

/*
 * //Algoritme 2V2CEM2 Oplossing=
 * Algo2v2CEM2.berekenAlgo2v2CEM2(aantalrondes, aantalnoten,
 * MidiElements, jSymVar, Oplossing, NotesMarkovTrans,
 * NotesMarkovTrans2);
 */

/*
 * //Algoritme 2v2CENM2 Oplossing=
 * Algo2v2CENM2.berekenAlgo2v2CENM2(aantalrondes, aantalnoten,
```

```
         * MidiElements, jSymVar, Oplossing, NotesMarkovTrans,
         * NotesMarkovTrans2);
         */

        // Algoritme 3
        Oplossing=
        Algo3.berekenAlgo3(aantalrondes, aantalnoten, MidiElements,
        jSymVar, NotesMarkovTrans, Oplossing, MarkovTrans,
        MarkovTrans2);

        // Oplossing schrijven
        MidiMaker.writeMidiFile(Oplossing);
        System.out.println("Oplossing: "+ Oplossing);

    }// einde SoloGenMain()

    // Main
    public static void main(String[] args){
        new SoloGenMain();
    }
}
```

Bijlage V.15: ZZ_MostCommonPitch

```
package soloGen;

import java.util.*;

public class ZZ_MostCommonPitch{

    public static int getMostCommonPitch(
    Vector<Integer> PedalPointVector){

        int most_common_pitch= 0;
        int tempcount5= 0;
        int count5= 0;

        // Berekenen meest voorkomende pitch
        for(int i= 0; i< PedalPointVector.size(); i++){
            for(int j= 0; j< PedalPointVector.size(); j++){
                if(i!= j&&
                Math.abs(PedalPointVector.elementAt(i))== Math
                .abs(PedalPointVector.elementAt(j))){
                    tempcount5++;
                }
            }
            if(tempcount5> count5){
                most_common_pitch=
                Math.abs(PedalPointVector.elementAt(i));
                count5= tempcount5;
            }
            tempcount5= 0;
        }

        return most_common_pitch;
    }

}
```

Bijlage VI: Het project ants

Bijlage VI.1: AntSystem

```
package ants;

import java.util.HashMap;
import java.util.Random;
import java.util.Vector;

/**
 * Ant System algoritme om een notensequentie te maken. Vanaf een
 * bepaalde 'node' (hier een noot) zijn de naburige nodes diegenen
 * waarvoor de Markov transitiekans niet 0 is. De afstand tussen deze
 * nodes worden bepaald door de (1/logx) functie van de
 * transitiekansen. De grootte van het feromonenspoor gerelateerd aan
 * een bepaalde oplossing wordt bepaald door de fit met de jSymbolic
 * waarden.
 *
 * @author Max
 *
 */

public class AntSystem{

    @SuppressWarnings("unchecked")
    public static
    Vector<Integer>
    makeAntSystem(int aantalnoten, Vector<Integer> MidiElements,
    HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans,
    Vector<Double> jSymVar, HashMap<Integer,Integer> ElementCountSolo){

        int ants= 20;
        int tmax= 100;
        double alpha= 0.1; // Intensityfactor
        double beta= 1- alpha; // Visibilityfactor
        double rho= 0.005;
        double elitaireparameter= 1000;

        HashMap<Integer,HashMap<Integer,Double>> TrailIntensity=
        new HashMap<Integer,HashMap<Integer,Double>>();
        HashMap<Integer,HashMap<Integer,Double>> TrailVisibility=
        new HashMap<Integer,HashMap<Integer,Double>>();
        HashMap<Integer,Double> InnerMapx1=
        new HashMap<Integer,Double>();
        HashMap<Integer,Double> InnerMapx2=
        new HashMap<Integer,Double>();
        HashMap<Integer,Double> InnerMapx3=
        new HashMap<Integer,Double>();

        Random random= new Random();
        int index= 0;
        int r= MidiElements.size();
        double rand= 0;

        Vector<Integer> Oplossing= new Vector<Integer>(aantalnoten);
        Vector<Vector<Integer>> OplVerzameling=
        new Vector<Vector<Integer>>(ants);
        for(int i= 0; i< ants; i++){
            OplVerzameling.add(i, Oplossing);
```

```
}
Vector<Double> DFWaardeVerzameling= new Vector<Double>(ants);
for(int i= 0; i< ants; i++){
    DFWaardeVerzameling.add((double) 999999999);
}
double doelfunctiewaarde= 0;

int firstnote= 0;
double denominator= 0;
double P= 0;
double globaltau= 0;
double localtau= 0;

// Initiële hashmaps Intensity vullen
for(int l= 0; l< MidiElements.size(); l++){
    for(int m= 0; m< MidiElements.size(); m++){
        InnerMapx1.put(MidiElements.elementAt(m),
        0.000000000001);
    }
    TrailIntensity.put(MidiElements.elementAt(l), InnerMapx1);
}

// Ant System algoritme

index= random.nextInt(r);
firstnote= MidiElements.elementAt(index);

for(int t= 0; t< tmax; t++){
    for(int k= 0; k< ants; k++){

        Oplossing.clear();
        Oplossing.add(0, firstnote);

        TrailVisibility=
        (HashMap<Integer,HashMap<Integer,Double>>) NotesMarkovTrans
        .clone();

        for(int i= 0; i< aantalnoten; i++){

            if(i> 0){

                InnerMapx1=
                TrailIntensity.get(Oplossing.elementAt(i- 1));
                InnerMapx2=
                TrailVisibility
                .get(Oplossing.elementAt(i- 1));

                // bereken de noemer van de formule
                denominator= 0;
                for(int l= 0; l< MidiElements.size(); l++){
                    denominator+=
                    Math
                    .pow(
                    InnerMapx1.get(MidiElements.elementAt(l)),
                    alpha)*
                    Math
                    .pow(
                    InnerMapx2.get(MidiElements.elementAt(l)),
                    beta);
```

```
                }

                // random proportional transition rule
                rand= Math.random();
                P= 0;

                for(int j= 0; j< MidiElements.size(); j++){
                    P+=
                    (Math
                    .pow(
                    InnerMapx1.get(MidiElements.elementAt(j)),
                    alpha)* Math
                    .pow(
                    InnerMapx2.get(MidiElements.elementAt(j)),
                    beta))/
                    denominator;

                    if(P>= rand){

                        Oplossing.add(i,
                        MidiElements.elementAt(j));
                        break;
                    }
                }
            }

        }// i

        // Bereken de doelfunctiewaarde en aanvaardt al dan
        // niet de nieuwe oplossing
        doelfunctiewaarde=
        DoelFunctie.berekenDoelFunctie(Oplossing, jSymVar,
        ElementCountSolo, MidiElements);

        if(doelfunctiewaarde< DFWaardeVerzameling
        .elementAt(k)){
            DFWaardeVerzameling.set(k, doelfunctiewaarde);
            OplVerzameling.set(k, Oplossing);
        }

    }// k

    // TrailIntensity updaten
    for(int k2= 0; k2< ants; k2++){

        Oplossing= OplVerzameling.elementAt(k2);

        // De 'elitaire parameter' bepaalt welk aandeel van de
        // mieren sporen mogen achterlaten
        if(DFWaardeVerzameling.elementAt(k2)< elitaireparameter*
        SmallestElement.getSmallest(DFWaardeVerzameling)){

            globaltau= 1/ DFWaardeVerzameling.elementAt(k2);

            for(int l2= 1; l2< Oplossing.size(); l2++){

                InnerMapx3=
                TrailIntensity
                .get(Oplossing.elementAt(l2- 1));
```

```
                    localtau=
                    InnerMapx3.get(Oplossing.elementAt(l2));

                    localtau+= globaltau;

                    InnerMapx3.put(Oplossing.elementAt(l2),
                    localtau);
                    TrailIntensity.put(
                    Oplossing.elementAt(l2- 1), InnerMapx3);
                }
            }

        }// k2

        // Sporen evaporeren
        for(int l= 0; l< MidiElements.size(); l++){

            InnerMapx3=
            TrailIntensity.get(MidiElements.elementAt(l));

            for(int m= 0; m< MidiElements.size(); m++){

                localtau=
                (1- rho)*
                InnerMapx3.get(MidiElements.elementAt(m));
                InnerMapx3.put(MidiElements.elementAt(m),
                localtau);
            }

            TrailIntensity.put(MidiElements.elementAt(l),
            InnerMapx3);
        }

        // System.out.println("t: "+t+" Doelfunctie waarden:
         "+DFWaardeVerzameling);
        // System.out.println("t: "+t+" Trail Intensity: "+TrailIntensity);
        System.out.println("t: "+ t+
        " kleinste doelfunctiewaarde: "+
        SmallestElement.getSmallest(DFWaardeVerzameling));

    }// t

    double minnie= 0;
    int dex= 0;
    minnie= SmallestElement.getSmallest(DFWaardeVerzameling);
    dex= DFWaardeVerzameling.indexOf(minnie);

    Oplossing= OplVerzameling.elementAt(dex);

    return Oplossing;
  }

}
```

Bijlage VI.2: AntSystemMain

```
package ants;

import java.util.HashMap;
import java.util.Vector;

/**
 *
 * @author Max
 *
 */

public class AntSystemMain{

    public AntSystemMain(){

        // Referentiesolo Dream Theater - Build Me Up, Break Me Down
        // @4min20
        int[] MidiSeq=
        new int[]{76, 80, 76, 73, 68, 64, 61, 56, 61, 64, 68, 73, 76,
        80, 76, 73, 68, 73, 76, 85, 80, 76, 73, 76, 80, 85, 80, 84,
        80, 81, 80, 78, 76, 75, 73, 72, 69, 66, 69, 72, 75, 78, 81,
        78, 81, 78, 75, 72, 69, 72, 75, 78, 81, 78, 84, 80, 81, 80,
        78, 80, 76, 78, 75, 76, 75, 73, 72, 68, 72, 75, 78, 76, 73,
        68, 64, 61, 64, 68, 73, 76, 73, 80, 73, 81, 73, 80, 73, 85,
        80, 76, 73, 68, 64, 68, 73, 76, 80, 85, 80, 76, 80, 73, 80,
        81, 78, 75, 72, 69, 72, 66, 69, 63, 66, 60, 63, 57, 60, 63,
        66, 69, 72, 75, 72, 68, 66, 68, 72, 75, 72, 68, 72, 75, 78,
        80, 84, 80, 78, 75, 78, 84, 87, 84, 80, 75, 80, 78};

        int aantalnoten= MidiSeq.length;

        // MidiSequentie en MidiElements
        Vector<Integer> MidiSequentie= new Vector<Integer>(0);
        Vector<Integer> MidiElements= new Vector<Integer>(0);

        for(int i= 0; i< MidiSeq.length; i++){
            MidiSequentie.add(MidiSeq[i]);
        }

        // Een vector vullen met de unieke elementen uit MidiSeq
        for(int i= 0; i< MidiSeq.length; i++){
            if(MidiElements.contains(MidiSeq[i])){}else{
                MidiElements.add(MidiSeq[i]);
            }
        }
        System.out.println("Grootte MidiElements: "+
        MidiElements.size());

        // Een hashmap maken die voor elk uniek element bewaart
        // hoeveel het voorkomt in de solo
        HashMap<Integer,Integer> ElementCountSolo=
        new HashMap<Integer,Integer>();
        ElementCountSolo= ElementCounter.counter(MidiSequentie);

        // NotesMarkovTrans
        HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();
        NotesMarkovTrans= NotesMarkovizer.makeMarkov(MidiSequentie);
```

```
        System.out.println("NotesMarkovTrans: "+ NotesMarkovTrans);

        // jSymVar
        Vector<Double> jSymVar= new Vector<Double>(0);
        jSymVar= JSymVar.getJSymVar(MidiSequentie);

        // Algoritme aanroepen
        Vector<Integer> Oplossing= new Vector<Integer>(0);
        Oplossing=
        AntSystem.makeAntSystem(aantalnoten, MidiElements,
        NotesMarkovTrans, jSymVar, ElementCountSolo);

        // Oplossing schrijven
        MidiMaker.writeMidiFile(Oplossing);
        System.out.println("Oplossing: "+ Oplossing);
    }

    public static void main(String[] args){
        new AntSystemMain();
    }

}
```

Bijlage VI.3: DoelFunctie

```
package ants;

import java.util.*;

/**
 * Berekent de waarde van de doelfunctie. Deze toetst de procentuele
 * afwijking tussen berekende eigenschappen van de gemaakte oplossing
 * en de respectievelijke referentiewaarden van deze variabelen.
 *
 * Bijkomend is er in de doelfunctie een soort van Cross Entropy term
 * gemaakt die een fout toevoegd gebaseerd op een Markov
 * transitiematrix.
 *
 * In deze doelfunctie is een Markov transitiematrix van zowel de
 * eerste als de tweede orde verwerkt.
 *
 * Bijkomend wordt de samenstelling van de referentiesolo vergeleken
 * met de gemaakte oplossing. Afwijkingen van de
 * referentiesamenstelling worden afgestrafd.
 *
 * @author Max
 *
 */

public class DoelFunctie{

    public static double berekenDoelFunctie(
    Vector<Integer> Oplossing, Vector<Double> jSymVar,
    HashMap<Integer,Integer> ElementCountSolo,
    Vector<Integer> MidiElements){

        // jSymbolic variabelen van de Oplossing berekenen
        Vector<Double> OplVar= new Vector<Double>(0);
        OplVar= JSymVar.getJSymVar(Oplossing);

        // Berekenen van de procentuele afwijkingen op de jSymbolic
        // gebaseerde referentiewaarden.
        double doelfunctiewaarde= 0;

        for(int i= 0; i< OplVar.size(); i++){

            if(jSymVar.elementAt(i)!= 0){
                doelfunctiewaarde+=
                (double) Math
                .abs((double) (OplVar.elementAt(i)- jSymVar
                .elementAt(i))/ jSymVar.elementAt(i));
            }
        }

        // Samenstelling noten van oplossing en referentiesolo moeten
        // ongeveer matchen
        int afwijking= 0;

        HashMap<Integer,Integer> ElementCountOpl=
        new HashMap<Integer,Integer>();
        ElementCountOpl= ElementCounter.counter(Oplossing);

        for(int i= 0; i< MidiElements.size(); i++){
```

```
            try{
                afwijking+=
                Math.abs(ElementCountSolo.get(MidiElements
                .elementAt(i))-
                ElementCountOpl.get(MidiElements.elementAt(i)));
            }catch(NullPointerException e){
                afwijking+=
                ElementCountSolo.get(MidiElements.elementAt(i));
            }

            doelfunctiewaarde+= 0.1* Math.pow(afwijking, 2);
            afwijking= 0;
        }

        return doelfunctiewaarde;

    }// einde method

}// einde class
```

Bijlage VI.4: ElementCounter

```
package ants;

import java.util.HashMap;
import java.util.Vector;

/**
 *
 * @author Max
 *
 */

public class ElementCounter{

    public static HashMap<Integer,Integer> counter(
    Vector<Integer> NootSequentie){

        HashMap<Integer,Integer> ElementCounts=
        new HashMap<Integer,Integer>();
        Vector<Integer> Elements= new Vector<Integer>(0);

        // Een vector vullen met de unieke elementen
        for(int i= 0; i< NootSequentie.size(); i++){

            if(Elements.contains(NootSequentie.elementAt(i))){}else{
                Elements.add(NootSequentie.elementAt(i));
            }
        }

        // HashMap vullen met de unieke elementen gelinkt aan het
        // aantal keer dat ze voorkomen
        int counter;
        for(int i= 0; i< Elements.size(); i++){

            counter= 0;

            for(int j= 0; j< NootSequentie.size(); j++){

                if(Elements.elementAt(i)== NootSequentie.elementAt(j)){
                    counter++;
                }
            }

            ElementCounts.put(Elements.elementAt(i), counter);
        }

        return ElementCounts;
    }

}
```

Bijlage VI.5: Intervals

```
package ants;

import java.util.*;

/**
 * berekent de vector met intervallen van een vector met noten
 *
 * @author Max
 *
 */
public class Intervals{

    public static Vector<Integer> getIntervals(
    Vector<Integer> PedalPointVector){

        Vector<Integer> Intervallen= new Vector<Integer>(0);

        for(int i= 1; i< PedalPointVector.size(); i++){

            Intervallen.add(PedalPointVector.elementAt(i)-
            PedalPointVector.elementAt(i- 1));
        }

        return Intervallen;
    }

}
```

Bijlage VI.6: JSymVar

```
package ants;

import java.util.Vector;

/**
 *
 * @author Max
 *
 */

public class JSymVar{

    public static Vector<Double> getJSymVar(
    Vector<Integer> MidiSequentie){

        Vector<Integer> Intervallen= new Vector<Integer>(0);
        Intervallen= Intervals.getIntervals(MidiSequentie);

        // Vector maken voor waarden van berekende variabelen
        // (jSymbolic gebaseerd)
        Vector<Double> jSymVar= new Vector<Double>(0);

        double arpeg=
        AmountOfArpeggiation.getAmountOfArpeggiation(Intervallen);
        jSymVar.add(arpeg);

        double avmi=
        AverageMelodicInterval.getAverageMelodicInterval(Intervallen);
        jSymVar.add(avmi);

        double chromes=
        ChromaticMotion.getChromaticMotion(Intervallen);
        jSymVar.add(chromes);

        double dm=
        DirectionOfMotion.getDirectionOfMotion(Intervallen);
        jSymVar.add(dm);

        double discommonint=
        DistanceBetweenMostCommonMelodicIntervals
        .getDistanceBetweenMostCommonMelodicIntervals(Intervallen);
        jSymVar.add(discommonint);

        double doma=
        DurationOfMelodicArcs.getDurationOfMelodicArcs(MidiSequentie,
        Intervallen);
        jSymVar.add(doma);

        double inbesp=
        IntervalBetweenStrongestPitches
        .getIntervalBetweenStrongestPitches(MidiSequentie);
        jSymVar.add(inbesp);

        double fifthz= MelodicFifths.getMelodicFifths(Intervallen);
        jSymVar.add(fifthz);

        double octavez= MelodicOctaves.getMelodicOctaves(Intervallen);
        jSymVar.add(octavez);
```

```
        double thirdz= MelodicThirds.getMelodicThirds(Intervallen);
        jSymVar.add(thirdz);

        double tritonez=
        MelodicTritones.getMelodicTritones(Intervallen);
        jSymVar.add(tritonez);

        double commelint=
        MostCommonMelodicInterval
        .getMostCommonMelodicInterval(Intervallen);
        jSymVar.add(commelint);

        double commelintprev=
        MostCommonMelodicIntervalPrevalence
        .getMostCommonMelodicIntervalPrevalence(Intervallen);
        jSymVar.add(commelintprev);

        double compitprev=
        MostCommonPitchPrevalence
        .getMostCommonPitchPrevalence(MidiSequentie);
        jSymVar.add(compitprev);

        double ncommint=
        NumberOfCommonMelodicIntervals
        .getNumberOfCommonMelodicIntervals(Intervallen);
        jSymVar.add(ncommint);

        double numcompit=
        NumberOfCommonPitches.getNumberOfCommonPitches(MidiSequentie);
        jSymVar.add(numcompit);

        double relstrcommint=
        RelativeStrengthOfMostCommonIntervals
        .getRelativeStrengthOfMostCommonIntervals(Intervallen);
        jSymVar.add(relstrcommint);

        double relstrtopitch=
        RelativeStrengthOfTopPitch
        .getRelativeStrengthOfTopPitch(MidiSequentie);
        jSymVar.add(relstrtopitch);

        double repnote= RepeatedNotes.getRepeatedNotes(Intervallen);
        jSymVar.add(repnote);

        double sizarcs=
        SizeOfMelodicArcs.getSizeOfMelodicArcs(Intervallen);
        jSymVar.add(sizarcs);

        double stepmo= StepwiseMotion.getStepwiseMotion(Intervallen);
        jSymVar.add(stepmo);

        return jSymVar;
    }

}
```

Bijlage VI.7: NotesMarkovizer

```
package ants;

import java.util.Collections;
import java.util.HashMap;
import java.util.Vector;

/**
 * Deze class maakt een Markov TransitieMatrix van een reeks gegeven
 * midinummers.
 *
 * @author Max
 *
 */

public class NotesMarkovizer{

    public static HashMap<Integer,HashMap<Integer,Double>>
    makeMarkov(Vector<Integer> MidiSequentie){

        HashMap<Integer,HashMap<Integer,Double>> NotesMarkovTrans=
        new HashMap<Integer,HashMap<Integer,Double>>();

        Vector<Integer> MidiElements= new Vector<Integer>(0);

        // Maakt verzameling van de unieke noten
        for(int i= 0; i< MidiSequentie.size(); i++){
            if(MidiElements.contains(MidiSequentie.elementAt(i))){}else{
                MidiElements.add(MidiSequentie.elementAt(i));
            }
        }

        Collections.sort(MidiElements);

        // Nootparen worden geteld en in een geneste hashmap gestoken
        int counter= 0;
        int jcounter= 0;
        HashMap<Integer,Integer> Rijtotaal=
        new HashMap<Integer,Integer>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){
                for(int k= 0; k< MidiSequentie.size()- 1; k++){
                    if(MidiElements.elementAt(i)== MidiSequentie
                    .elementAt(k)&&
                    MidiElements.elementAt(j)== MidiSequentie
                    .elementAt(k+ 1)){
                        counter++;
                    }
                }// k

                MarkovTrans2.put(MidiElements.elementAt(j),
                (double) counter);
                jcounter+= counter;
                counter= 0;
```

```
            }// j

            NotesMarkovTrans.put(MidiElements.elementAt(i),
            MarkovTrans2);

            if(jcounter== 0){
                Rijtotaal.put(MidiElements.elementAt(i), 1);
            }else{
                Rijtotaal.put(MidiElements.elementAt(i), jcounter);
            }// hashmap van rijtotalen om transitiematrix te kunnen
             // maken

            jcounter= 0;
        }// i

        // De hashmap met getelde nootparen wordt per rij gedeeld door
        // de respectievelijke rijtotalen
        // teneinde een transitiematrix te bekomen
        Double absel;
        double relel= 0;
        HashMap<Integer,Double> InnerMap=
        new HashMap<Integer,Double>();

        for(int i= 0; i< MidiElements.size(); i++){

            HashMap<Integer,Double> MarkovTrans2=
            new HashMap<Integer,Double>();

            for(int j= 0; j< MidiElements.size(); j++){

                InnerMap=
                NotesMarkovTrans.get(MidiElements.elementAt(i));
                absel= InnerMap.get(MidiElements.elementAt(j));
                relel=
                absel/ Rijtotaal.get(MidiElements.elementAt(i));
                MarkovTrans2.put(MidiElements.elementAt(j), relel);
            }

            NotesMarkovTrans.put(MidiElements.elementAt(i),
            MarkovTrans2);
        }

        return NotesMarkovTrans;
    }// makeMarkov

}
```

Bijlage VI.8: SmallestElement

```
package ants;

import java.util.*;

/**
 * Gets the smallest element of a vector of type double
 *
 * @author Max
 *
 */

@SuppressWarnings("unused") public class SmallestElement{

    public static double getSmallest(java.util.Vector<Double> Vector){

        double smallest= 999999999;
        for(int i= 0; i< Vector.size(); i++){

            if(Vector.elementAt(i)< smallest){
                smallest= Vector.elementAt(i);
            }
        }

        return smallest;
    }
}
```